

ALMA MATER STUDIORUM – UNIVERSITA' DI BOLOGNA

SECONDA FACOLTA' DI INGEGNERIA CON SEDE A CESENA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Graph&SODA: A Graphical Tool for SODA

Tesi di Laurea in
Sistemi Distribuiti L-A

Relatore

Ill.mo Prof. Ing. Andrea Omicini

Presentata da

Luca Staccoli

Correlatore

Dott. Ing. Ambra Molesini

Sessione Terza

Anno Accademico 2008/2009

*A mio nonno Guerrino,
alla mia Famiglia e al mio Amore*

Ringrazio innanzitutto il Prof. Ing. Andrea Omicini e la Dott. Ing. Ambra Molesini per la professionalità, la disponibilità e la pazienza con cui mi hanno seguito e guidato durante questo percorso.

Ringrazio la mia famiglia e tutte le persone che mi hanno sostenuto con il loro affetto e la loro vicinanza in questi anni.

Indice

1.	Introduzione.....	9
2.	SODA.....	11
2.1	Introduzione.....	11
2.2	Il meta-modello Agenti & Artefatti.....	12
2.3	Le fasi del processo SODA.....	16
2.3.1	Layering.....	19
2.3.2	Fase di Analisi.....	21
2.3.2.1	Analisi dei Requisiti.....	22
2.3.2.2	Dall'Analisi dei Requisiti all'Analisi.....	23
2.3.2.3	Analisi.....	25
2.3.3	Fase di Design.....	27
2.3.3.1	Dall'Analisi al Design Architetturale.....	27
2.3.3.2	Design Architetturale.....	28
2.3.3.3	Dal Design Architetturale al Design di Dettaglio (Carving).....	32
2.3.3.4	Design di Dettaglio.....	35
3.	Eclipse.....	40
3.1	Introduzione.....	40
3.2	Struttura della piattaforma.....	41
3.2.1	Workbench.....	42
3.2.2	Workspace.....	43
3.2.3	Help.....	43
3.2.4	Team.....	44
3.3	Utilizzo ed espansione della piattaforma.....	44
3.3.1	I Plugin di Eclipse.....	45
3.3.2	Creazione di Plugin in Eclipse.....	49
4.	Analisi del sistema.....	60
4.1	Specifiche del sistema.....	60
4.2	Analisi dei Requisiti del sistema.....	61

4.3	Casi d'uso e scenari.....	66
4.3.1	Creazione delle entità astratte.....	67
4.3.2	Gestione delle relazioni tra le varie entità astratte.....	67
4.3.3	Inserimento tabelle già precompilate.....	67
4.3.4	Layering.....	68
4.3.5	Carving.....	68
4.3.6	Esportazione tabelle.....	69
4.3.7	Vista delle entità astratte (Abstractions View).....	69
4.3.8	Vista delle tabelle (Tables View).....	69
4.3.9	Visualizzazione delle tabelle di cui fa parte una specifica entità.....	69
4.3.10	Integrazione con altri plugin.....	70
4.4	Schema logico di riferimento.....	70
5.	Progettazione ed Implementazione del sistema.....	73
5.1	Struttura delle entità.....	73
5.1.1	Astrazione.....	74
5.1.2	Tabella.....	76
5.1.3	Layer.....	77
5.2	Le viste.....	78
5.2.1	Abstractions View.....	79
5.2.1.1	Layering.....	80
5.2.1.2	Carving.....	81
5.2.2	Tables View.....	81
5.3	Implementazione del sistema.....	83
6.	Conclusioni e sviluppi futuri.....	84
7.	Bibliografia.....	85

1. Introduzione

Nel mondo d'oggi un ingegnere del software si trova a dover progettare sistemi che vanno ad operare in ambienti sempre più complessi, quali *Internet*; questi, infatti, richiedono controllo, dinamicità, interagibilità ed espandibilità, aumentando così le difficoltà della loro gestione.

Per questi ed altri motivi, da anni sta emergendo la necessità di orientare la progettazione verso gli “agenti” [1]: per la gestione dei suddetti ambienti, viene preferito il concetto di *agente*, in quanto entità pro-attiva ed autonoma, piuttosto di quello di oggetto, inteso principalmente come entità “passiva” e funzionale.

Assieme al concetto di agente, ne vengono introdotti altri (quali *l'ambiente* percepito dagli agenti e le loro *interazioni*) che portano all'approfondimento dell'Ingegneria del Software Orientata agli Agenti (*Agent-Oriented Software Engineering, AOSE*).

Tale approfondimento ha portato alla nascita di diverse metodologie orientate agli agenti, mostrando tecnologie per lo sviluppo del processo di progettazione del sistema. Una di queste, sviluppata presso l'Ateneo dell'Università di Bologna, è SODA (*Societies in Open and Distributed Agent spaces*), che si occupa principalmente dell'interazione tra le varie entità presenti, dei ruoli che gli agenti hanno all'interno del sistema da loro percepito, e soprattutto della introduzione di nuove entità, gli *artefatti* [2], che verranno usate dagli agenti del sistema.

Dalla sua creazione nel 2001 [3], grazie ai ricercatori del DEIS (Dipartimento di Elettronica, Informatica e Sistemistica) dell'Università di Bologna, SODA ha subito varie trasformazioni e migliorie (dai meccanismi utilizzati alle fasi del suo processo), fino ad arrivare alla versione a cui si farà riferimento in questa sede.

Per facilitare l'utilizzo di tale metodologia, si vuole realizzare un

tool grafico (*Graph&SODA*) che assista il progettista nello sviluppo di un sistema attraverso le varie fasi della metodologia SODA. Tale realizzazione grafica viene eseguita attraverso la creazione di un plugin Eclipse, piattaforma open-source estendibile per lo sviluppo, interfacciamento ed esecuzione di software applicativi.

La tesi si svilupperà nel seguente modo:

- Capitolo 2: presentazione della metodologia SODA e descrizione del suo processo.
- Capitolo 3: presentazione e descrizione della piattaforma Eclipse e della sua struttura.
- Capitolo 4: analisi del sistema e suoi casi d'uso.
- Capitolo 5: progettazione ed implementazione del sistema.
- Capitolo 6: conclusioni e sviluppi futuri.
- Capitolo 7: bibliografia.

2. SODA

2.1 Introduzione

Con i continui sviluppi e le innovazioni nel campo dell'Ingegneria del Software (*Software Engineering* - SE), si è sentita la necessità di un diverso approccio rispetto alla tradizionale SE: l'aumento della complessità dei sistemi software ha portato allo sviluppo di Sistemi Multi-Agente (*Multi-Agent Systems* – MAS) concentrato sull'astrazione dell'agente (*agent*), in quanto entità in grado di raggiungere il proprio scopo (*goal*) eseguendo azioni sia richieste da altre entità (se presenti in una società), sia autonomamente (differenza con i sistemi ad oggetti, *object-oriented*), attraverso interazioni di protocolli e linguaggi ad alto livello. Ma non solo: nell'Ingegneria del Software orientata agli agenti (*Agent-Oriented Software Engineering* – AOSE), sono altrettanto importanti l'ambiente in cui essi lavorano (*environment*) e le relazioni presenti tra i vari agenti che possono anche formare società (*society*).

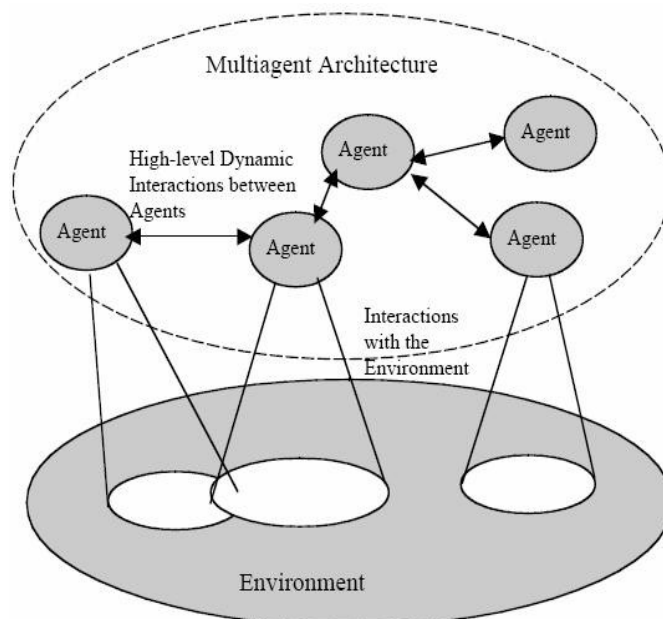


Figura 2.1: Architettura MAS

Tale nuova architettura (figura 2.1) è ben diversa da quella tradizionale ad oggetti, ove questi erano entità funzionali, orientate a fornire un servizio e tutto era portato ad essere rappresentato (appunto) da un oggetto: il “presunto” controllo di esecuzione nei sistemi *object-oriented* (l'esecuzione di un processo o di un limitato set di processi sincronizzati) è tuttavia impensabile per sistemi dinamici più complessi come i MAS, nei quali il controllo di esecuzione viene assegnato ai componenti attivi del sistema stesso (gli agenti), mentre le loro relazioni ed anche quelle tra i componenti passivi (risorse dell'ambiente, *environmental resources*) vengono stabilite a priori, dando una maggior consistenza e visione del sistema.

Tra le varie metodologie presenti che analizzano e progettano i MAS, in questo capitolo ci si occuperà di SODA (*Societies in Open and Distributed Agent spaces*), che si basa sul meta-modello Agenti & Artefatti (*Agents & Artifacts meta-model* – A&A): dalle caratteristiche delle astrazioni adottate da tale meta-modello, conseguirà la stesura di tabelle relazionali (delle quali si parlerà più avanti).

2.2 Il meta-modello Agenti & Artefatti

Nel meta-modello A&A [4] le principali astrazioni sono gli agenti (*agents*), gli artefatti (*artifacts*) e l'ambiente di lavoro (*workspaces*). Gli agenti sono i componenti attivi che svolgono un lavoro autonomo all'interno dell'ambiente (come descritto in precedenza). Gli artefatti sono invece la parte funzionale del MAS: possono essere utilizzati e manipolati dagli agenti per raggiungere al meglio il proprio goal (*individual artifacts*) o per il relazionarsi tra gli agenti stessi (*social artifacts*), possono anche fare da tramite tra gli agenti e le risorse

esterne permettendo la loro visione all'interno del MAS e quindi il loro utilizzo (*environmental artifacts*). Tali artefatti, affinché possano essere utilizzati dagli agenti, devono disporre delle seguenti caratteristiche:

- Interfaccia d'uso (*usage interface*): definisce l'elenco delle operazioni che un agente può scegliere di far eseguire all'artefatto stesso. Questa è la differenza con gli agenti: gli agenti non necessitano di un'interfaccia d'uso in quanto sono autonomi, al contrario degli artefatti che sono controllati dagli agenti e che eseguono un servizio richiesto in modo prevedibile.
- Istruzioni operative (*operating instructions*): corrispondono alla descrizione della procedura che un agente deve seguire per interagire con l'artefatto, praticamente un manuale di istruzioni.
- Descrizione funzionale (*function description*): è un'astrazione dell'implementazione dell'artefatto, in quanto descrive la sua funzionalità ovvero che cosa si può ottenere dall'artefatto stesso.

Oltre a tali caratteristiche, un artefatto deve avere anche altre proprietà atte a “monitorare” l'artefatto stesso nonché a far migliorare il loro utilizzo da parte degli agenti. Eccole elencate qui a seguito:

- Ispezionabilità (*Inspectability*): rende disponibile agli agenti tutte o parte delle caratteristiche dell'artefatto (stato, contenuto, istruzioni operative, descrizione funzionale) al fine di capire e ispezionare le dinamiche e il comportamento del

MAS.

- Controllabilità (*Controllability*): è un'estensione dell'ispezionabilità, dando la possibilità di controllare e monitorare il corretto funzionamento dell'artefatto anche a tempo di esecuzione.
- Malleabilità (*Malleability*): collegata all'ispezionabilità, permette di cambiare il comportamento dell'artefatto a tempo di esecuzione per adattarlo meglio al MAS di cui fa parte.
- Predicibilità (*Predictability*): permette all'agente di prevedere il comportamento dell'artefatto, grazie alle caratteristiche date dall'artefatto stesso e viste così come una sorta di contratto tra agente e artefatto.
- Formalizzazione (*Formalisability*): collegata alla predicibilità, permette alla precisa descrizione delle caratteristiche di essere rappresentata in un preciso modello semantico.
- Collegabilità (*Linkability*): è la capacità di un artefatto ad invocare operazioni su altri artefatti, dando l'opportunità quindi di creare nuovi artefatti in modo incrementale da altri già esistenti.
- Distribuzione (*Distribution*): attraverso alla *linkability*, rende possibile la creazione di un singolo artefatto distribuito, composto da più artefatti singoli anche appartenenti a reti diverse ma collegati tra loro.

Esistono vari tipi di artefatti all'interno di un MAS, ognuno dei quali fornisce diversi servizi per gli agenti ed costituendo diversi modelli,

tecnologie e strumenti per adattarsi ad un vasta gamma di applicazioni. Risulta utile ed opportuno quindi catalogare i vari tipi di artefatti in tre categorie:

- Artefatti individuali (*Individual artifacts*): utilizzati solo da un agente (anche se attraverso la *linkability* esegue operazioni richieste da altri artefatti all'interno del MAS), interponendosi tra l'agente stesso e l'ambiente.
- Artefatti sociali (*Social artifacts*): a differenza di quelli individuali, sono utilizzati da due o più agenti all'interno del MAS, intermediando tra loro e l'ambiente e mirando al goal della società piuttosto di quello individuale.
- Artefatti ambientali (*Environmental artifacts*): incapsulano le risorse esterne, permettendo agli agenti del MAS di utilizzarle attraverso interazioni ad alto livello.

Tali artefatti (individuali, sociali e ambientali) costituiscono la “colla” dei MAS, permettendo le relazioni tra gli agenti e la strutturazione dell'ambiente di lavoro (*workspace*) ove agenti ed artefatti esistono. I *workspaces* possono essere definiti come topologia dell'ambiente di lavoro; ne possono esistere molteplici, ricordando sempre che agenti e artefatti possono interagire tra loro solo se appartenenti allo stesso *workspace*, anche se l'intersezione tra più *workspaces* è possibile grazie allo spostamento dinamico (entrata o uscita) degli agenti e alla presenza dello stesso artefatto su più *workspaces* (sempre con la possibilità di essere aggiunto o cavato dinamicamente).

2.3 Le fasi del processo SODA

Come detto in precedenza, la metodologia SODA (il cui meta-modello è riportato in figura 2.3) adotta, tra le altre, le principali entità concettuali provenienti dal meta-modello A&A descritto nella sezione precedente.

Tale metodologia si cura delle relazioni presenti tra i vari componenti del MAS attraverso la stesura di tabelle relazionali, risolvendo così la complessità della rappresentazioni dei componenti di un MAS ed introducendo un principio di livelli di astrazione (*layering*, di cui ne parleremo nel paragrafo successivo) per semplificare la visione del sistema MAS stesso. Queste tabelle relazionali sono identificate attraverso un nome univoco ed un acronimo, che specificano quali entità sono relazionate e a quale livello di astrazione appartengono. Ad esempio la prima tabella che incontriamo è $(L)AR_t$, che rappresenta la tabella (*table*, specificata dalla “*t*” in pedice) di relazione tra un attore (*Actor*, specificato dalla “*A*”) e un requisito (*Requirement*, specificato dalla “*R*”) al livello di astrazione *L* (*Layer*, rappresentato tra parentesi tonde). Le vedremo in dettaglio nei prossimi paragrafi.

SODA è strutturata in due fasi, Analisi e Design [5], che a loro volta sono costituiti da due sotto-fasi, rispettivamente Analisi dei Requisiti e Analisi per la prima e Design Architettuale e Design di Dettaglio per la seconda (figura 2.2).

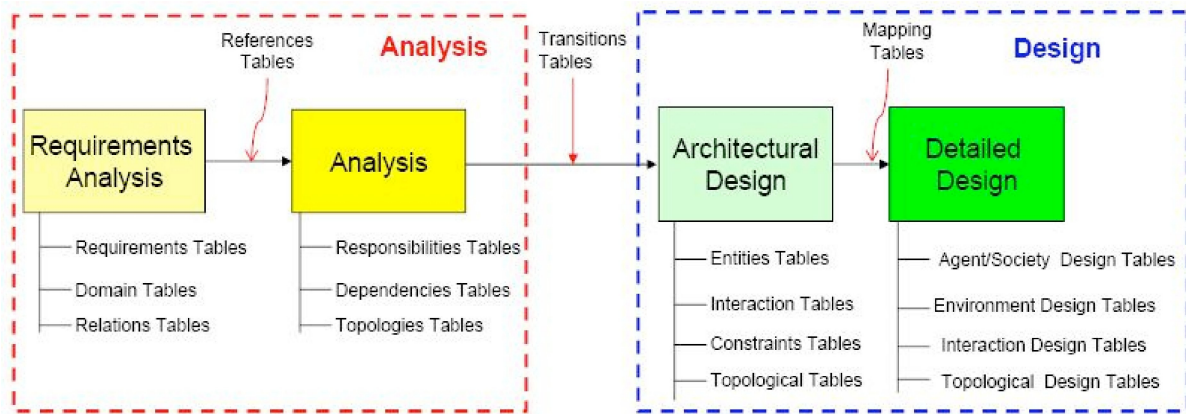


Figura 2.2: Fasi del processo SODA

- **Analisi dei requisiti (*Requirements Analysis*):** vengono introdotte diverse entità astratte, tra le quali: gli attori (*actor*, quelli che generano richieste dall'esterno), i loro requisiti (*requirement*), l'ambiente esterno (*external environment*) che contiene i sistemi esterni già presenti (*legacy systems*), e le relazioni che sono presenti tra tutte queste astrazioni (*relation*).
- **Analisi (*Analysis*):** le entità astratte presentate nell'analisi dei requisiti vengono ora espresse in entità più concrete, ovvero in *tasks* (attività che richiedono una o più competenze) e funzioni (*function*, attività reattive che sono da supporto ai *tasks*). Le relazioni tra le varie entità astratte sono l'inizio delle loro dipendenze (*dependency*, ad esempio interazioni, vincoli, ecc.). La struttura dell'ambiente è modellata in termini di topologie (*topology*) ovvero vincoli sull'ambiente. Le topologie spesso derivano dalle funzioni, ma possono anche influire sul conseguimento dei *tasks*.

- Design Architettuale (*Architectural Design*): in questa fase le responsabilità di conseguire i *task* vengono date ai ruoli (*roles*), i quali saranno in grado di eseguire azioni (*actions*), e le responsabilità di fornire le funzioni vengono date alle risorse (*resources*), le quali saranno in grado di eseguire operazioni (*operations*) fornendo una o più funzioni. Le dipendenze diventano interazioni (*interactions*) ma anche regole (*rules*) che coinvolgono il comportamento delle entità (ruoli e risorse). Infine dalle topologie si passa alla definizione di spazi (*spaces*), ovvero spazi concettuali che strutturano l'ambiente.
- Design di Dettaglio (*Detailed Design*): questa fase è espressa in termini di agenti (*agents*, entità autonome che rappresentano diversi ruoli), società di agenti (*agent societies*, ovvero gruppi di agenti ed artefatti che nell'insieme hanno un comportamento autonomo), artefatti (*artifacts*, adeguate entità funzionali che mappano le risorse, vedi paragrafo 2.2) ed aggregati (*aggregates*, ovvero gruppi di agenti ed artefatti che nell'insieme hanno un comportamento funzionale, reattivo). Ora gli spazi diventano gli ambienti di lavoro (*workspaces*), dove sono presenti set di agenti ed artefatti, i quali rispettivamente possono entrare od uscire ed essere aggiunti o tolti.

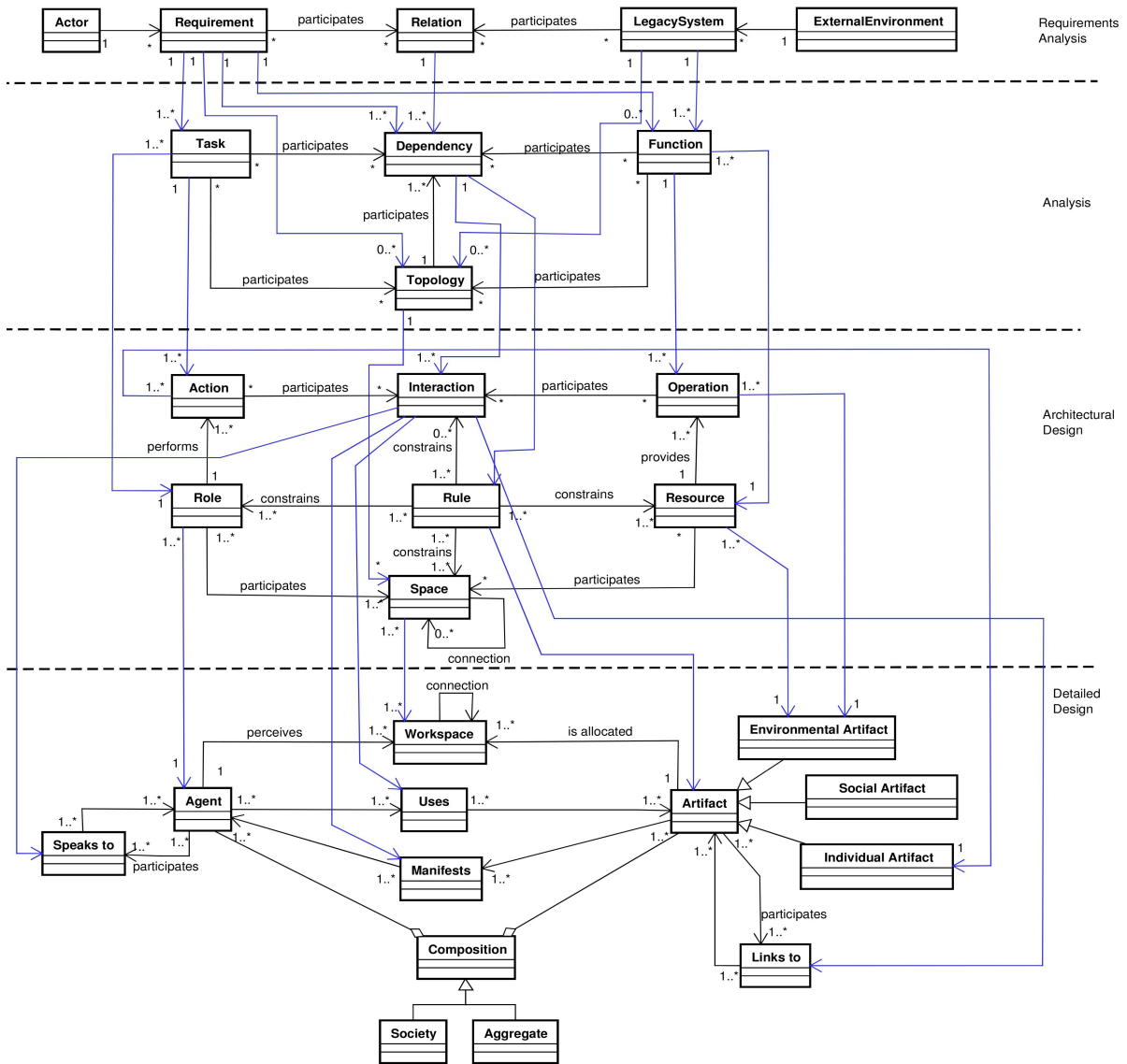


Figura 2.3: Meta-modello di SODA

2.3.1 Layering

Come detto in precedenza, per gestire al meglio la complessità di rappresentazione di un MAS si introduce in SODA un semplice principio di layering [6]: il sistema è rappresentato come composizione di agenti ed artefatti in diversi livelli di astrazione (*layers of abstraction*) più o meno dettagliati ed indipendenti ma

relazionabili tra loro solo attraverso operazioni di *layering*. Tali operazioni sono due: *zoom* e *proiezione* (*projection*). Lo zoom rende possibile il passaggio da un livello di astrazione ad un altro più o meno dettagliato: in particolare l'operazione di *in-zoom* permette di zoomare le entità dal loro livello (più astratto) ad un livello più dettagliato, mentre l'operazione di *out-zoom* fa l'opposto, mostra le entità dal loro livello (più dettagliato) ad un livello più astratto. Se invece non si ha la necessità di zoomare su un'entità bensì di utilizzarla in livelli di astrazione più o meno dettagliati così com'è, si utilizza la proiezione.

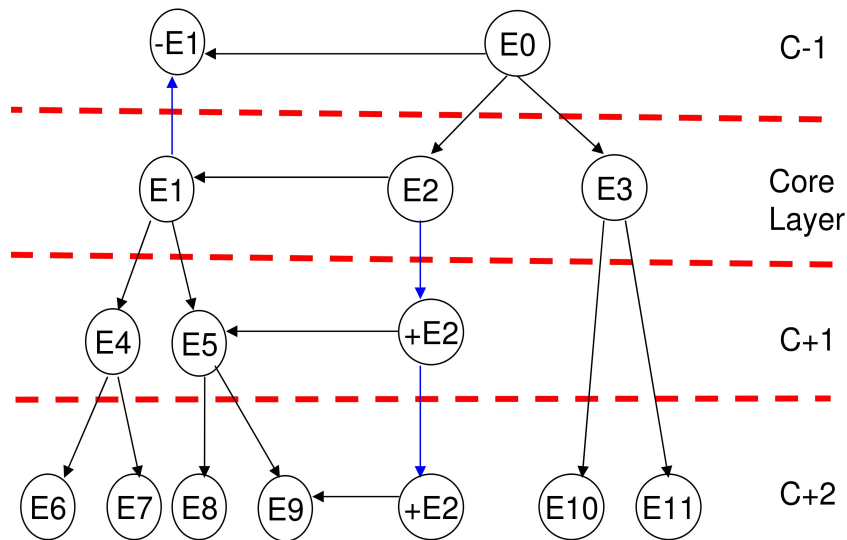


Figura 2.4: Layers in SODA

Il *layer* di partenza in SODA si chiama *core layer* (rappresentato dalla lettera “C”), il quale deve essere *completo*, ovvero che contiene tutte le entità richieste per descrivere appieno il livello di astrazione dato (vedi figura 2.4). Gli altri livelli sono frutto delle operazioni di *layering*, diventando così non necessariamente completi. Questi ultimi vengono denominati “C+1” (o “C+2” e via dicendo) i livelli

che hanno subito l'operazione in-zoom dal livello “C” (o “C+1” e così via), mentre l'operazione inversa, out-zoom, vengono denominati “C-1” (o “C-2” ecc.). Le entità proiettate vengono rappresentate attraverso l'anteposizione di un “+” se proiettate in un livello più dettagliato, viceversa con un “-” se proiettate in un livello più astratto.

Il principio di *layering* può essere utilizzato nelle fasi di Analisi dei Requisiti, Analisi e Design Architettuale; non può essere utilizzato in quella di Design di Dettaglio poiché per principio non è più possibile aumentare il dettaglio. Il principio di *layering* viene rappresentato anch'esso attraverso la tabella di Zoom (*Zooming Table* $(L)Z_t$, figura 2.5), mettendo in relazione un *layer* più astratto con uno più dettagliato (sia con l'*in-zoom* sia con l'*out-zoom*).

Layer L	Layer L+1
out-zoomed entity	in-zoomed entities

Figura 2.5: Tabella di Zoom $(L)Z_t$

2.3.2 Fase di Analisi

In questo paragrafo verrà sviluppata la fase di Analisi: in particolare verranno illustrate le tabelle della sotto-fase di Analisi dei Requisiti (paragrafo 2.3.2.1), le tabelle del passaggio dalla sotto-fase di Analisi dei Requisiti a quella di Analisi (paragrafo 2.3.2.2) e le tabelle della sotto-fase di Analisi (paragrafo 2.3.2.3).

2.3.2.1 Analisi dei Requisiti

In questa sotto-fase (come detto nel paragrafo 2.3) ci si occupa di illustrare le tabelle che descriveranno gli attori, i loro requisiti, l'ambiente esterno e le relazioni che intercorrono tra essi. In particolare sono presenti tre tipologie di tabelle:

- Tabelle dei Requisiti (*Requirements Tables*, figura 2.6): si occupano delle entità astratte che trattano i requisiti, in particolare una tabella che descrive gli attori (*Actor table*, $(L)Ac_t$), un'altra i requisiti di ciascuno di essi (*Actor-Requirement table*, $(L)AR_t$) e un'altra ancora la descrizione dei singoli requisiti (*Requirement table*, $(L)Re_t$).

Actor	Description
<i>actor name</i>	<i>actor description</i>

Actor	Requirement
<i>actor name</i>	<i>requirement names</i>

Requirement	Description
<i>requirement name</i>	<i>requirement description</i>

Figura 2.6: Tabelle dei Requisiti, $(L)Ac_t$, $(L)AR_t$, $(L)Re_t$

- Tabelle di Dominio (*Domain Tables*, figura 2.7): si occupano delle entità astratte che riguardano l'ambiente esterno, in particolare una tabella che mostra i *legacy systems* associati a ciascun ambiente esterno (*ExternalEnvironment-LegacySystem table*, $(L)EELS_t$) e un'altra la descrizione dei singoli *legacy systems* (*LegacySystem table*, $(L)LS_t$).

External-Environment	Legacy-System
<i>external-environment name</i>	<i>legacy-system names</i>

Legacy-System	Description
<i>legacy-system name</i>	<i>legacy-system description</i>

Figura 2.7: Tabelle di Dominio, $(L)EELS_t$, $(L)LS_t$

- Tabelle delle Relazioni (*Relations Tables*, figura 2.8): si occupano delle relazioni che intercorrono tra le varie entità astratte, in particolare una tabella che descrive tutte le relazioni (*Relation table*, $(L)Rel_t$), un'altra le relazioni in cui è coinvolto ciascun requisito (*Requirement-Relation table*, $(L)RR_t$) e un'altra ancora le relazioni in cui è coinvolto ciascun legacy system (*LegacySystem-Relation table*, $(L)LSR_t$).

Relation	Description
<i>relation name</i>	<i>relation description</i>

Requirement	Relation
<i>requirement name</i>	<i>relation names</i>

Legacy-System	Relation
<i>legacy-system name</i>	<i>relation names</i>

Figura 2.8: Tabelle delle Relazioni, $(L)Rel_t$, $(L)RR_t$, $(L)LSR_t$

2.3.2.2 Dall'Analisi dei Requisiti all'Analisi

Per il passaggio dalla sotto-fase di Analisi dei Requisiti a quella di Analisi, vengono introdotte le tabelle dei riferimenti (*References*

Tables, figura 2.9) che identificano le relazioni tra le entità astratte delle due sotto-fasi. In particolare una tabella che si occupa delle relazioni tra ogni requisito e i *task* generati (*Reference Requirement-Task table (L)RRT_i*), una tabella che si occupa delle relazioni tra ogni requisito e le funzioni generate (*Reference Requirement-Function table (L)RRF_i*), una tabella che si occupa delle relazioni tra ogni *legacy-system* e le corrispondenti funzioni (*Reference LegacySystem-Function table (L)RLSF_i*), una tabella che si occupa delle relazioni tra i *legacy-system* e le topologie (*Reference LegacySystem-Topology table (L)RLST_i*), una tabella che associa le dipendenze alle relazioni (*Reference Relation-Dependency table (L)RRD_i*).

Requirement	Task
<i>requirement name</i>	<i>task names</i>

Requirement	Function
<i>requirement name</i>	<i>function names</i>

Legacy-System	Function
<i>legacy-system name</i>	<i>function names</i>

Legacy-System	Topology
<i>legacy-system name</i>	<i>topology names</i>

Relation	Dependency
<i>relation name</i>	<i>dependency names</i>

Figura 2.9: Tabelle delle Riferimenti dall'Analisi dei Requisiti all'Analisi, (L)RRT_i, (L)RRF_i, (L)RLSF_i, (L)RLST_i, (L)RRD_i

2.3.2.3 Analisi

In questa sotto-fase vengono illustrate le tabelle che descriveranno i *task*, le funzioni (generate non solo dai requisiti, ma anche dai *legacy-system* che potrebbero anche costituire vincoli topologici, come visto in figura 2.9), le dipendenze (interazioni, vincoli, ecc.) derivate dalle relazioni, le topologie. In particolare sono presenti tre tipologie di tabelle:

- Tabelle delle Responsabilità (*Responsabilities Tables*, figura 2.10): si occupano di descrivere le entità astratte legate al concetto di responsabilità, in particolare una tabella descrive i task (*Task table* $(L)T_i$) e un'altra tabella descrive le funzioni (*Function table* $(L)F_i$).

Task	Description
<i>task name</i>	<i>task description</i>

Function	Description
<i>function name</i>	<i>function description</i>

Figura 2.10: Tabelle delle Responsabilità, $(L)T_i$, $(L)F_i$

- Tabelle delle Dipendenze (*Dependencies Tables*, figura 2.11): si occupano delle dipendenze che intercorrono tra le varie entità astratte, in particolare una tabella che si occupa di tutte le dipendenze (*Dependency table* $(L)D_i$), una tabella che si occupa delle dipendenze in cui un task è coinvolto (*Task-Dependency table* $(L)TD_i$), una tabella che si occupa delle dipendenze in cui una funzione è coinvolta (*Function-Dependency table* $(L)FD_i$), una tabella che si occupa delle

dipendenze che coinvolgono le tipologie (*Topology-Dependency table* $(L)TopD_t$).

Dependency	Description
<i>dependency name</i>	<i>dependency description</i>

Task	Dependency
<i>task name</i>	<i>dependency names</i>

Function	Dependency
<i>function name</i>	<i>dependency names</i>

Topology	Dependency
<i>topology name</i>	<i>dependency names</i>

Figura 2.11: Tabelle delle Dipendenze,
 $(L)D_t$, $(L)TD_t$, $(L)FD_t$, $(L)TopD_t$

- Tabelle delle Topologie (*Topologies Tables*, figura 2.12): si occupano dei vincoli topologici sull'ambiente, in particolare una tabella che descrive tutti i vincoli topologici (*Topology table* $(L)Top_t$), un'altra tabella che si occupa dei vincoli topologici riguardanti i task (*Task-Topology table* $(L)TTop_t$), e un'altra tabella ancora che si occupa dei vincoli topologici riguardanti le funzioni (*Function-Topology table* $(L)FTop_t$).

Topology	Description
<i>topology name</i>	<i>topology description</i>

Task	Topology
<i>function name</i>	<i>topology names</i>

Function	Topology
<i>function name</i>	<i>topology names</i>

Figura 2.12: Tabelle delle Topologie, $(L)Top_i$, $(L)TTop_i$, $(L)FTop_i$

2.3.3 Fase di Design

In questo paragrafo verrà presentata la fase di Design: in particolare verranno illustrate le tabelle del passaggio dalla sotto-fase di Analisi a quella di Design Architetturale (paragrafo 2.3.3.1), le tabelle della sotto-fase di Design Architetturale (paragrafo 2.3.3.2), le tabelle del passaggio dalla sotto-fase di Design Architetturale a quella di Design di Dettaglio illustrando il meccanismo di *carving* (paragrafo 2.3.3.3) e le tabelle della sotto-fase di Design di Dettaglio (paragrafo 2.3.3.4).

2.3.3.1 Dall'Analisi al Design Architetturale

Per il passaggio dalla sotto-fase di Analisi a quella di Design Architetturale, vengono introdotte le tabelle delle transizioni (*Transitions Tables*, figura 2.13) che identificano le relazioni tra le entità astratte delle due sotto-fasi. In particolare una tabella che si occupa delle relazioni tra ogni ruolo e i relativi *task* (*Transition Role-Task table* $(L)TRT_i$), una tabella che si occupa dell'assegnazione delle azioni ad ogni *task* (*Transition Task-Action table* $(L)TTA_i$), una tabella che si occupa delle relazioni tra ogni risorsa e le funzioni associate (*Transition Resource-Function table* $(L)TRF_i$), una tabella che si occupa dell'assegnazione delle operazioni ad ogni funzione (*Transition Function-Operation table* $(L)TFO_i$), una tabella che associa le dipendenze alle interazioni

(*Transition Interaction-Dependency table* $(L)TID_t$), una tabella che associa le dipendenze alle regole (*Transition Rule-Dependency table* $(L)TRuD_t$), una tabella che associa gli spazi alle topologie (*Transition Topology-Space table* $(L)TTopS_t$).

Role	Task
<i>role name</i>	<i>task names</i>

Task	Action
<i>task name</i>	<i>action names</i>

Resource	Function
<i>resource name</i>	<i>function names</i>

Function	Operation
<i>function name</i>	<i>operation names</i>

Interaction	Dependency
<i>interaction name</i>	<i>dependency names</i>

Rule	Dependency
<i>rule name</i>	<i>dependency names</i>

Topology	Space
<i>topology name</i>	<i>spaces names</i>

Figura 2.13: Tabelle delle Transizioni dall'Analisi al Design Architettuale, $(L)TRT_t$, $(L)TTA_t$, $(L)TRF_t$, $(L)TFO_t$, $(L)TID_t$, $(L)TRuD_t$, $(L)TTopS_t$

2.3.3.2 Design Architettuale

In questa sotto-fase vengono illustrate le tabelle che descriveranno i

ruoli (che sono in grado di eseguire una o più azioni), le risorse (che sono in grado di fornire una o più operazioni), le interazioni e le regole tra le varie entità astratte (derivanti dalle dipendenze della sotto-fase precedente), gli spazi (dalle topologie). In particolare sono presenti quattro tipologie di tabelle:

- Tabelle delle Entità (*Entities Tables*, figura 2.14): si occupano di descrivere i ruoli (con le loro azioni) e le risorse (con le loro operazioni), in particolare una tabella che descrive le azioni disponibili di un ruolo (*Action Table (L)A_i*), una tabella che descrive le operazioni disponibili di una risorsa (*Operation Table (L)O_i*), una tabella che specifica quali azioni può compiere ciascun ruolo (*Role-Action Table (L)RA_i*), una tabella che specifica quali operazioni può fornire ciascuna risorsa (*Resource-Operation Table (L)RO_i*).

Action	Description
<i>action name</i>	<i>action description</i>

Operation	Description
<i>operation name</i>	<i>operation description</i>

Role	Action
<i>role name</i>	<i>action names</i>

Resources	Operation
<i>resource name</i>	<i>operation names</i>

Figura 2.14: Tabelle delle Entità, (L)A_i, (L)O_i, (L)RA_i, (L)RO_i

- Tabelle delle Interazioni (*Interaction Tables*, figura 2.15): si occupano delle interazioni che coinvolgono le azioni dei ruoli

e le operazioni delle risorse, in particolare una tabella che descrive tutte le interazioni (*Interaction Table* $(L)I_t$), una tabella che mostra le interazioni coinvolgenti le azioni (*Action-Interaction Table* $(L)AcI_t$), una tabella che mostra le interazioni coinvolgenti le operazioni (*Operation-Interaction Table* $(L)OpI_t$).

Interaction	Description
<i>interaction name</i>	<i>interaction description</i>

Action	Interaction
<i>action name</i>	<i>interaction names</i>

Operation	Interaction
<i>operation name</i>	<i>interaction names</i>

Figura 2.15: Tabelle delle Interazioni, $(L)I_t$, $(L)AcI_t$, $(L)OpI_t$

- Tabelle dei Vincoli (*Constraints Tables*, figura 2.16): si occupano di mostrare i vincoli rilevati che coinvolgono le varie entità astratte, in particolare una tabella che descrive le singole regole (*Rule Table* $(L)Ru_t$), una tabella che mostra i vincoli sulle interazioni (*Interaction-Rule Table* $(L)IRu_t$), una tabella che mostra le regole coinvolgenti le risorse (*Resource-Rule Table* $(L)ReRu_t$), una tabella che mostra le regole coinvolgenti i ruoli (*Role-Rule Table* $(L)RoRu_t$), una tabella che mostra le regole coinvolgenti gli spazi (*Space-Rule Table* $(L)SRu_t$).

Rule	Description
<i>rule name</i>	<i>rule description</i>

Interaction	Rule
<i>interaction name</i>	<i>rule names</i>

Resource	Rule
<i>resource name</i>	<i>rule names</i>

Role	Rule
<i>role name</i>	<i>rule names</i>

Space	Rule
<i>space name</i>	<i>rule names</i>

Figura 2.16: Tabelle delle Interazioni, $(L)I_t$, $(L)AcI_t$, $(L)OpI_t$

- Tabelle Topologiche (*Topological Tables*, figura 2.17): si occupano di descrivere la struttura logica dell'ambiente ove risiedono tutte le entità, in particolare una tabella che descrive gli spazi (*Space Table* $(L)S_t$), una tabella che mostra le connessioni tra i vari spazi allo stesso livello di astrazione (*Space-Connection Table* $(L)SC_t$), una tabella che mostra la presenza di ogni risorsa in determinati spazi (*Space-Resource Table* $(L)SRe_t$), potendo così offrire un servizio distribuito su più nodi della rete, ed infine una tabella che mostra gli spazi percepiti da ogni ruolo (*Space-Role Table* $(L)SRo_t$).

Space	Description
<i>space name</i>	<i>space description</i>

Space	Connection
<i>space name</i>	<i>space names</i>

Resource	Space
<i>resource name</i>	<i>space names</i>

Role	Space
<i>role name</i>	<i>space names</i>

Figura 2.17: Tabelle Topologiche, $(L)S_t$, $(L)SC_t$, $(L)SRe_t$, $(L)SRo_t$

2.3.3.3 Dal Design Architettureale al Design di Dettaglio (Carving)

Il compito dell'ultima sotto-fase è proprio di scegliere il livello di rappresentazione più adeguato per ciascuna entità architettureale, portando alla descrizione di un design di dettaglio tra le diverse alternative possibili. Tale meccanismo viene chiamato *carving*. Ad esempio, considerando la figura 2.18 (sezione sinistra), si ipotizza che nella fase di Design Architettureale si abbia un sistema costituito dai ruoli $R1$ e $R2$ nel *core layer C*, dai ruoli $R4$, $R5$ e la proiezione di $R2$ nel livello $C+1$, e dai ruoli $R6$, $R7$, $R8$ e $R9$ nel livello $C+2$. Di qui si comincia ad eseguire il meccanismo di *carving*, ovvero la scelta di un livello di rappresentazione (zoom o proiezione) di ciascuna entità: partendo dal *core layer C* si decide, ad esempio, di zoomare solo $R1$, lasciando $R2$ al livello base di rappresentazione (il *core layer C* appunto); passando al livello $C+1$ si decide di zoomare ulteriormente $R4$, considerando così anche $R6$ e $R7$ al livello $C+2$; mentre per $R5$ si decide di lasciarlo così com'è al livello $C+1$. Si ottiene così la scelta dei ruoli compresi nella linea viola (figura 2.18, sezione centrale).

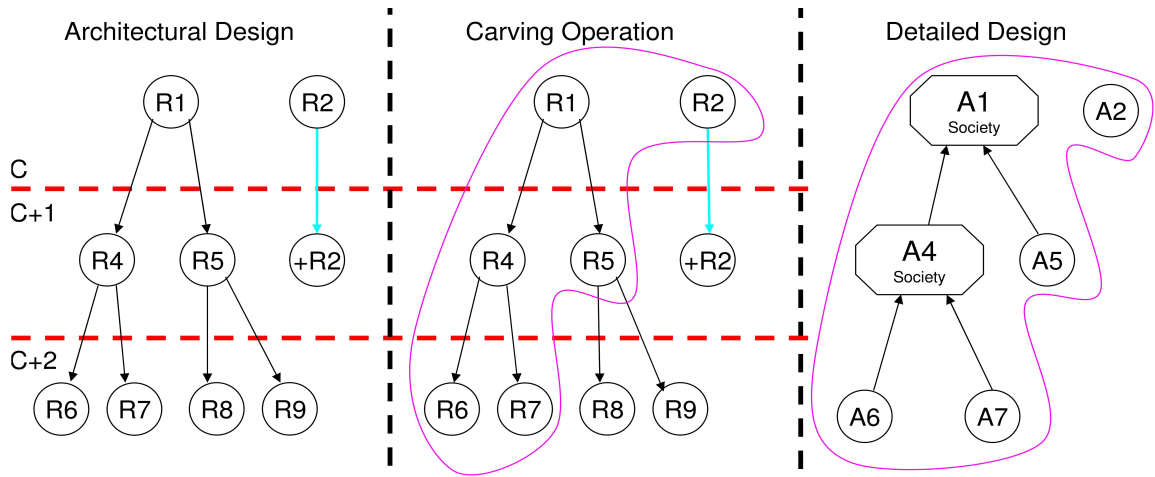


Figura 2.18: Fase di Design e meccanismo di *carving*

Il risultato di tale meccanismo è il passaggio dalle entità espresse nella sotto-fase di Design Architetturelle a quelle nella sotto-fase di Design di Dettaglio. In particolare, nel caso in figura 2.18 (sezione destra), *R1* andrà mappato in una società di agenti (*A1*), mentre *R2* andrà mappato in un agente individuale (*A2*); *A1* è composto da due entità, *R4* e *R5*: *R4* (essendo stato zumato nel *carving*) andrà mappato in un'altra società di agenti (*A4*), mentre *R5* andrà mappato in un agente individuale (*A5*); infine *R6* e *R7* verranno mappati in agenti individuali (rispettivamente in *A6* ed in *A7*). Come risultato si ha la stesura delle tabelle di *Mapping* (*Mapping Tables*, figura 2.19), derivate dalla scelta dei vari livelli di rappresentazione; in particolare una tabella che mappa i ruoli in agenti (*Mapping Agent-Role table MAR_i*), una tabella che mappa i ruoli in società di agenti, avendo deciso di zumarli (*Mapping Society-Role table MSR_i*), una tabella che mappa le azioni in artefatti individuali (*Mapping Artifact-Action table MAA_i*), una tabella che mappa le risorse in artefatti ambientali (*Mapping Artifact-Resource table MArR_i*), una tabella che mappa le risorse in aggregati, avendo deciso di zumarle (*Mapping Aggregate-Resource table MAggR_i*), una tabella che mappa

le operazioni in artefatti ambientali (*Mapping Artifact-Operation table MArOp_i*), una tabella che mappa le regole precedentemente trovate in artefatti che le implementano facendole rispettare (*Mapping Artifact-Rule table MArRu_i*), una tabella che mappa gli spazi in *workspaces* (*Mapping Space-Workspace table MSW_i*), una tabella che mappa le interazioni in usi (*Mapping Interaction-Uses table MIU_i*), una tabella che mappa le interazioni in manifestazioni (*Mapping Interaction-Manifests table MIM_i*), una tabella che mappa le interazioni in *speaksTo*, ovvero in relazionabilità con altri agenti (*Mapping Interaction-SpeaksTo table MIS_i*), una tabella che mappa le interazioni in *linksTo*, ovvero in collegabilità con altri artefatti (*Mapping Interaction-LinksTo table MIL_i*).

Agent	Role
<i>agent name</i>	<i>role names</i>

Society	Role
<i>society name</i>	<i>role name</i>

(Individual) Artifact	Action
<i>artifact name</i>	<i>action names</i>

(Environmental) Artifact	Resource
<i>artifact name</i>	<i>resource names</i>

Aggregate	Resource
<i>aggregate name</i>	<i>resource name</i>

(Environmental) Artifact	Operation
<i>artifact name</i>	<i>operation names</i>

Artifact	Rule
<i>agent name</i>	<i>rule names</i>

Space	Workspace
<i>space name</i>	<i>workspace name</i>

Interaction	Uses
<i>interaction name</i>	<i>uses name</i>

Interaction	Manifests
<i>interaction name</i>	<i>manifests name</i>

Interaction	Speaks To
<i>interaction name</i>	<i>speaksTo name</i>

Interaction	Links To
<i>interaction name</i>	<i>linksTo name</i>

Figura 2.19: Tabelle di *Mapping* dal Design Architeturale al Design di Dettaglio, MAR_t , MSR_t , MAA_t , MAR_t , $MAggr_t$, $MArOp_t$, $MARu_t$, MSW_t , MIU_t , MIM_t , MIS_t , MIL_t

2.3.3.4 Design di Dettaglio

In questa ultima sotto-fase, come già accennato nelle tabelle di *Mapping* precedentemente, vengono introdotti gli agenti, le loro società, gli artefatti, gli aggregati (composizione di agenti ed artefatti con comportamento funzionale) e i *workspaces* (collegandosi tra loro e rappresentando, appunto, i diversi ambienti di lavoro di tutte le entità). Vengono introdotte quattro tipologie di tabelle:

- Tabelle di Progetto di Agenti e Società (*Agent/Society Design Tables*, figura 2.20): si occupano di descrivere le relazioni tra tali entità, in particolare una tabella che mostra quali artefatti (individuali) sono associati a ciascun agente (*Agent-Artifact*

$table AA_t$), una tabella che mostra quali agenti fanno parte di ciascuna società (*Society-Agent table SA_t*), una tabella che mostra quali artefatti (principalmente sociali) sono associati a ciascuna società di agenti (*Society-Artifact table Sar_t*).

Agent	(Individual) Artifact
<i>agent name</i>	<i>artifact names</i>

Society	Agent
<i>society name</i>	<i>agent names</i>

Society	Artifact
<i>society name</i>	<i>artifact names</i>

Figura 2.20: Tabelle di Progetto di Agenti e Società,
 AA_t , SA_t , SAr_t

- Tabelle di Progetto dell'Ambiente (*Environment Design Tables*, figura 2.21): si occupano di descrivere le relazioni tra le restanti entità presenti nell'ambiente, in particolare una tabella che mostra le operazioni di cui dispone ciascun artefatto artefatto (*Artifact-UserInterface table AUI_t*), una tabella che mostra quali artefatti appartengono a ciascun aggregato (*Aggregate-Artifact table AggAr_t*), una tabella che mostra quali agenti appartengono a ciascun aggregato (*Aggregate-Agent table AggAge_t*).

Artifact	User Interface
<i>artifact name</i>	<i>userInterface names</i>

Aggregate	Artifact
<i>aggregate name</i>	<i>artifact names</i>

Aggregate	Agent
<i>aggregate name</i>	<i>agent names</i>

Figura 2.21: Tabelle di Progetto dell'Ambiente,
 AUI_i , $AggAr_i$, $AggAge_i$

- Tabelle di Progetto delle Interazioni (*Interaction Design Tables*, figura 2.22): si occupano di descrivere la progettazione delle varie interazioni tra tutte le entità, in particolare una tabella che descrive il protocollo per ogni interazione *uses* (*Uses-Protocol table* UP_i), una tabella che mostra le interazioni di tipo *uses* e gli agenti che ne sono coinvolti (*Uses-Agent table* $UAge_i$), una tabella che descrive il protocollo per ogni interazione *speaksTo* (*SpeaksTo-Protocol table* SP_i), una tabella che mostra le interazioni di tipo *speaksTo* e gli agenti che ne sono coinvolti (*SpeaksTo-Agent table* $SpAge_i$), una tabella che descrive il protocollo per ogni interazione di tipo *manifests* (*Manifests-Protocol table* MP_i), una tabella che mostra le interazioni di tipo *manifests* e gli artefatti che ne sono coinvolti (*Manifests-Artifact table* $MArt_i$), una tabella che descrive il protocollo per ogni interazione di tipo *linksTo* (*LinksTo-Protocol table* LP_i), una tabella che mostra le interazioni di tipo *linksTo* e gli artefatti che ne sono coinvolti (*LinksTo-Artifact table* $LArt_i$).

Uses	Protocol
<i>uses name</i>	<i>protocol description</i>

Uses	Agent
<i>uses name</i>	<i>agent names</i>

Speaks To	Protocol
<i>speaksTo name</i>	<i>protocol description</i>

Speaks To	Agent
<i>speaksTo name</i>	<i>agent names</i>

Manifests	Protocol
<i>manifests name</i>	<i>protocol description</i>

Manifests	Artifact
<i>manifests name</i>	<i>artifact names</i>

Links To	Protocol
<i>linksTo name</i>	<i>protocol description</i>

Links To	Artifact
<i>linksTo name</i>	<i>artifact names</i>

Figura 2.22: Tabelle di Progetto delle Interazioni,
 UP_t , $UAge_t$, SP_t , $SpAge_t$, MP_t , $MArt_t$, LP_t , $LArt_t$.

- Tabelle di Progetto Topologico (*Topological Design Tables*, figura 2.23): si occupano la struttura dettagliata dell'ambiente di lavoro, in particolare una tabella che descrive i *workspaces* (*Workspace table* W_t), una tabella che mostra le connessioni tra i vari *workspaces* (*Workspace-Connection table* WC_t), una tabella che mostra in quali *workspaces* sono allocati gli artefatti (*Workspace-Artifact table* $WArt_t$), una tabella che mostra quali *workspaces* può percepire ciascun agente (*Workspace-Agent table* WA_t).

Workspace	Description
<i>workspace name</i>	<i>workspace description</i>

Workspace	Connection
<i>workspace name</i>	<i>workspace names</i>

Workspace	Artifact
<i>workspace name</i>	<i>artifact names</i>

Agent	Workspace
<i>agent name</i>	<i>workspace names</i>

Figura 2.23: Tabelle di Progetto delle Interazioni,
 W_t , WC_t , $WArt_t$, Wa_t .

3. Eclipse

3.1 Introduzione

E' difficile stabilire cosa sia esattamente Eclipse; ognuno gli attribuisce un significato diverso [7]:

- “E' un ambiente libero di sviluppo Java.”
- “E' un ambiente flessibile per sperimentare nuovi linguaggi del computer o estensioni ai linguaggi esistenti.”
- “E' una struttura comprensiva che sviluppa diverse avanzate e moderne tecniche di progettazione di software e di implementazione.”

Eclipse non è chiaramente solo questo. E' un ambiente di sviluppo integrato (*Integrated Development Environment, IDE*) che dà la possibilità di gestire ambienti di lavoro, di costruire, lanciare ed eliminare gli errori di applicazioni, di personalizzare e condividere esperienze di programmazione.

Si può quindi definire Eclipse una piattaforma di sviluppo (in quanto non è un'applicazione autonoma finita), progettata per essere estesa indefinitamente mediante sofisticati sotto-sistemi, i plugin (di cui se ne parlerà più avanti).

Il progetto di Eclipse è stato creato nel Novembre del 2001 da un gruppo di società industriali (quali Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft and Webgain), che nel 2004, dopo il rilascio da parte di IBM della piattaforma Eclipse come Open Source, si riorganizzò in un'associazione no-profit, chiamata Eclipse Foundation. Tale

progetto permette liberamente lo sviluppo di software anche ad utenti finali, promuovendo così l'ampliamento della comunità di Eclipse: tutte le tecnologie e i codici sorgenti forniti e sviluppati da questa comunità, sono resi disponibili attraverso la licenza pubblica di Eclipse (*Eclipse Public License*).

3.2 Struttura della piattaforma

Eclipse è scritto nel linguaggio Java [8], rendendolo così eseguibile per i diversi sistemi operativi che lo supportano (ad esempio Linux, Mac OS X, Windows).

La piattaforma Eclipse [9] (figura 3.1) si basa sull'interazione tra i sotto-sistemi che la formano: questi sono implementati in uno o più plugin, che possono essere realizzati e modificati dall'utente finale. La struttura di tali plugin è costruita sopra la piattaforma *Runtime*, ovvero quella che si occupa dei plugin a tempo di esecuzione. I sotto-sistemi che costituiscono la piattaforma Eclipse sono: *Workbench*, *Workspace*, *Help*, *Team*.

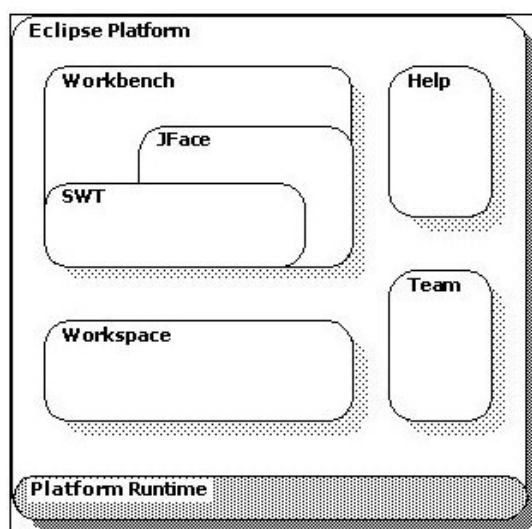


Figura 3.1: vista generale della piattaforma Eclipse

3.2.1 Workbench

Il *Workbench* (vedi paragrafo 3.3.1) è l'insieme dei plugin che contribuiscono a formare l'ambiente di sviluppo, avendo come obiettivo la creazione, la gestione e la navigazione delle risorse dell'ambiente di lavoro.

La sua interfaccia grafica si basa su SWT (*Standard Widget Toolkit*), che è un insieme di librerie contenenti le interfacce della programmazione di applicazioni (*Application Programming Interface, API*), molto semplificate ed “alleggerite” per permettere così l'implementazione e l'uso nelle diverse piattaforme. Un altro interfacciamento grafico lo si ha usando *JFace*, che è appunto una struttura di applicazioni Java basata sempre su SWT, ma che permette di creare più facilmente un'applicazione con interfaccia grafica basata su Java.

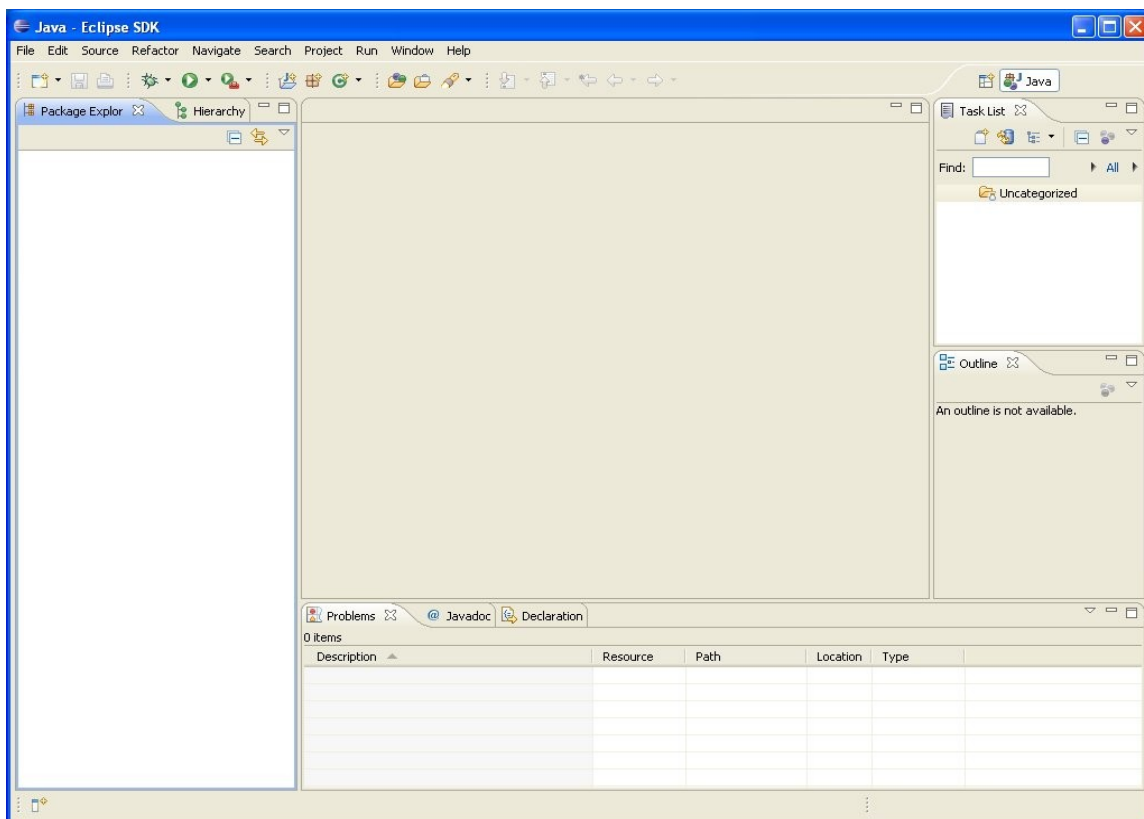


Figura 3.2: Java perspective

Esistono diverse vedute d'insieme (*perspective*) del *Workbench*, che possono essere anche personalizzate: quelle principali, nonché predefinite, sono quelle di Java (figura 3.2), di sviluppo di plugin, delle risorse e tante altre. Tali vedute sono composte principalmente da viste (*views*) e da modificatori (*editors*): le *views* sono quelle aree dedicate alla visualizzazione del progetto e di qualsiasi altra attività presente nel *Workbench*, mentre gli *editors* sono aree in cui si permette la modifica dei file scelti.

Altri componenti delle vedute sono le barre degli strumenti (*toolbars*), gli appunti (*markers*), i segnalibri (*bookmarks*), ecc.; tutti possono essere personalizzati e spostati in qualsiasi parte del *Workbench*, creando appunto diverse vedute.

3.2.2 Workspace

Con *Workspace* (vedi paragrafo 3.3.1) si intende lo spazio di lavoro in cui si sta operando; in particolare ci si riferisce alla cartella, allocata del filesystem, in cui sono archiviati tutti i file che fanno parte dei progetti a cui si sta lavorando: proprio per questo motivo, all'apertura del programma, viene richiesto di immettere il *Workspace*, così da poter riprendere i progetti già iniziati per poterli riutilizzare in seguito.

3.2.3 Help

Come dice il nome stesso (aiuto), *Help* assiste (se richiamato) l'utente nello sviluppo del suo progetto, dando informazioni ed esempi d'uso dell'operazione che si sta eseguendo. Proprio per questo è un componente necessario ed essenziale per l'utilizzo delle

varie operazioni messe a disposizione dal *Workbench*.

3.2.4 Team

Un altro componente importante è *Team* (squadra): questa area viene utilizzata per la creazione di progetti condivisi con altri swaristi, riuscendo a lavorarci e a modificarli in contemporanea, stabilendo le parti di cui ogni componente del team si deve occupare e mantenendo la loro visione e la loro sincronizzazione anche remota.

3.3 Utilizzo ed espansione della piattaforma

Eclipse è caratterizzato dall'uso e dall'interazione tra i vari plugin scelti ed utilizzati; la piattaforma, infatti, viene resa disponibile allo scaricamento in varie configurazioni: la versione base di Eclipse è utilizzabile solo come editor di file, vi si possono creare progetti generici e condividerli con una comunità, avendo praticamente la funzione di un esploratore di risorse del file-system. Altre versioni rilasciate, comprendono già plugin atti a sviluppare progetti in linguaggio Java (*Java Development Tools, JDT*), in C++ (*C/C++ Development Tools, CDT*) e altri ancora a sviluppare i plugin stessi (*Plugin Development Environment, PDE*). Attraverso l'utilizzo di tutti questi plugin, è possibile passare dall'ambiente integrato di Eclipse ad una applicazione perfezionata e completa che non sembra essere un IDE.

3.3.1 I Plugin di Eclipse

Tutto in Eclipse può essere visto come plugin: i principali che permettono il funzionamento della piattaforma sono denominati *org.eclipse.core*, per specificare che sono plugin indispensabili, creati dalla comunità di Eclipse per la sua esecuzione. Esempi di questi plugin sono:

- *org.eclipse.core.resources* (rappresentante il *Workspace*);
- *org.eclipse.ui* (rappresentante il *Workbench*);
- *org.eclipse.core.runtime* (rappresentante la piattaforma *Runtime*).

Un plugin è l'incapsulamento di comportamenti e dati che interagisce con altri plugin per formare un programma eseguibile; può essere visto come un oggetto nella programmazione orientata agli oggetti. Al suo interno possiede parti di codice che verranno compilate e poi eseguite solo alla invocazione del plugin stesso. Viene anche specificato quali altri plugin (o *Java packages*) sono necessari per la sua esecuzione, oltre ad esportare e a rendere disponibili i propri *Java packages*. Un plugin può essere anche esportato come directory o come file *JAR*, dando così la possibilità di essere utilizzato da altri plugin o applicazioni (insieme di plugin). Un'altra delle caratteristiche principali dei plugin di Eclipse, infatti, è proprio l'estendibilità: ognuno di essi dispone di due componenti, gli *extension point* e le *extension*. I primi si occupano di fornire l'interfacciamento (tipicamente una combinazione di XML ed interfacce Java) che un plugin esterno deve implementare per il collegamento col plugin destinatario, attraverso le *extension*, che, al contrario, si occupano dell'interfacciamento di un plugin con uno

esterno, ovvero implementano le interfacce date dai plugin esterni per essere collegati con le funzionalità del plugin stesso, senza essere a conoscenza di cosa sia il plugin chiamato, tranne lo scopo del contratto previsto, appunto, dall'*extension point*. Un esempio pratico è dato dalle prese di corrente: le prese rappresentano gli *extension point*, mentre le spine, le *extension*. Questa caratteristica è una delle motivazioni che spinge i progettisti ad utilizzare tale piattaforma (come viene fatto anche in questa sede), in quanto si ha già una base di partenza (l'IDE appunto) e la possibilità di utilizzare componenti che altri utenti hanno sviluppato in precedenza, estendendo, in seguito, la piattaforma stessa con plugin propri.

Un componente chiave di un plugin è il file *plugin.xml* (*plugin manifest file*): scritto in XML, esso contiene i suoi dettagli più importanti, quali il suo nome, l'ID (proprio identificativo) e il numero della sua versione. Altre informazioni sono opzionali, ad esempio quale codice Java fornisce e quali sono i plugin necessari per la sua esecuzione; un plugin, infatti, può fornire del codice, oppure solamente la documentazione, insiemi di risorse o altri dati utilizzabili da altri plugin. Possono anche essere specificati le *extension* e gli *extension point* se il plugin permette sue estensioni e/o collegamenti con altri plugin.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin id="com.xyz.myplugin" name="My Plugin" class="com.xyz.MyPlugin" version="1.0">
  <runtime>
    <library name="MyPlugin.jar"/>
  </runtime>
  <requires>
    <import plugin="org.eclipse.core.runtime"/>
  </requires>
</plugin>
```

Figura 3.3: Esempio di un manifest plugin.xml

Considerando l'esempio in figura 3.3, si può notare la struttura di tale file: il primo tag indica la versione di XML e la codifica dei caratteri utilizzati; il secondo specifica la versione di Eclipse per cui è stato creato questo plugin; il terzo mostra le informazioni base del plugin (ID, nome, versione), includendo, in questo caso, l'attributo “*class*” che specifica una istanza della classe di *Plugin* associata a questo; siccome contiene una sotto-classe di *Plugin*, questo plugin deve includere un elemento *runtime* che specifica il file *JAR* contenente il codice ed un elemento *requires* per importare il plugin *org.eclipse.core.runtime* dove risiede la classe importata. Il *manifest file* viene analizzato dalla piattaforma al momento del caricamento del plugin, esaminando tutte le informazioni necessarie per mostrarlo nell'interfaccia-utente (*User Interface*, *UI*), ad esempio icone, oggetti del menù, barre degli strumenti. Il codice di implementazione, presente in un file Java *JAR* separato, è caricato solamente quando il plugin viene richiamato e deve essere eseguito. Per questo motivo, tale procedimento viene considerato come “caricamento pigro” (*lazy loading*) dei plugin, ma rende possibile la semplicità e la “leggerezza” nell'esecuzione della piattaforma. Questa dinamicità dell'utilizzo dei plugin è stata introdotta nella versione 3.0 di Eclipse, attraverso l'incorporazione nel kernel di un altro *framework* di componenti Java: *OSGi* [10] (*Open Services Gateway initiative*). Questo è un sistema di moduli dinamici per Java, la cui piattaforma di servizio fornisce le funzioni per cambiare la composizione di un sistema dinamicamente, senza la necessità di riavvii. Il cuore di tale tecnologia è l'*OSGi Framework* (figura 3.4).

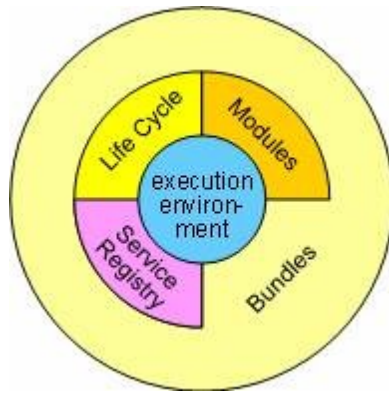


Figura 3.4: OSGi Framework

L'*OSGi Framework* fornisce un ambiente standardizzato per le applicazioni (chiamate *Bundles*, “insiemi”). Esso si suddivide in quattro livelli, dal quale Eclipse eredita le loro caratteristiche principali:

- Livello 0, *Execution Environment*: sono le specifiche dell'ambiente Java, ovvero le configurazioni e i profili di Java (ad esempio J2SE, CDC, CLDC, MIDP, ecc.).
- Livello 1, *Modules*: definisce le politiche per il caricamento delle classi (gestendo la loro visibilità in base alla loro tipologia, private o pubbliche), integrandosi completamente con l'architettura di sicurezza.
- Livello 2, *Life Cycle Management*: permette di aggiungere i *bundles* che possono essere inseriti, iniziati, arrestati, aggiornati e rimossi dinamicamente; i *bundles* si appoggiano sul Livello 1 per il caricamento delle classi, ma qui aggiungono un API per gestire i moduli a tempo di esecuzione. Anche questo livello è protetto dall'architettura di sicurezza, rendendo impossibile l'attacco da virus.

- Livello 3, *Service Registry*: fornisce un modello di cooperazione per i *bundles* che prende in considerazione le dinamiche, ovvero un modello completo per condividere dinamicamente gli oggetti tra i *bundles* stessi, a tempo di esecuzione. Il modello dei servizi tiene conto anche della sicurezza tra le comunicazioni dei vari *bundles*.

Ereditando le sue proprietà, Eclipse acquista nuove caratteristiche alla sua struttura, in particolare l'aggiunta e la rimozione dinamica dei plugin e un modello di sicurezza più robusto, dando la possibilità di eseguire più istanze della piattaforma lavorando su medesimi o differenti *Workspaces*.

3.3.2 Creazione di Plugin in Eclipse

Il modo più “semplice” per creare un plugin in Eclipse è utilizzare il *Plugin Development Environment (PDE)*, il quale, appunto, è un'estensione (ovvero un plugin) per Eclipse che gestisce ed assiste l'utente nella creazione di un plugin, collaborando con il *Java Development Tooling (JDT)*, in quanto dovrà trattare con codice Java.

Nella figura 3.5 viene data una veduta d'insieme di come un plugin creato è collegato al resto della piattaforma.

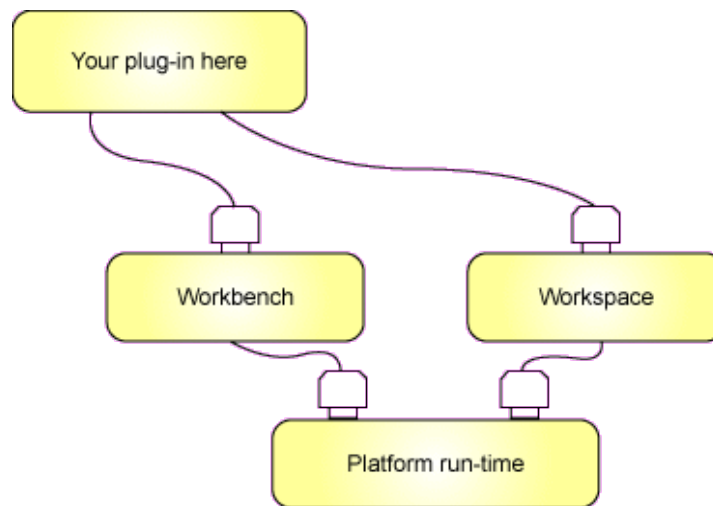


Figura 3.5: Veduta d'insieme dei principali collegamenti dei plugin creati con la piattaforma

Il *PDE* possiede una procedura guidata durante la creazione di plugin, offrendo la possibilità di partire da un “progetto modello”, ad esempio una *view*, un *editor*, ecc.: seguendo passo a passo le informazioni necessarie per la loro creazione, questo procedimento semplifica il lavoro, in quanto ogni passo “automatico” in più che si fa, è una parte di codice Java che viene scritta automaticamente dalla collaborazione tra il *PDE* e il *JDT*, lasciando sempre la possibilità di andarlo a modificare in seguito manualmente.

Per rendere più pratici la spiegazione e l'apprendimento del suo utilizzo, viene riportato qui di seguito la creazione tramite il *PDE* di un semplice plugin che, selezionando nel menù **Hello!-> Display Hello World!**, mostrerà una finestra contenente la scritta “Hello World!”.

Come prima cosa si apre Eclipse, specificando il *Workspace* dove si vuole lavorare. Si deve selezionare la *perspective* riguardante il *Plugin Development*, andando nel menù **Window->Open**

Perspective->Other... e selezionare **Plugin Development**.

A questo punto si può iniziare un nuovo progetto, andando nel menù **File->New->Project->Other**, poi selezionando **Plugin Project**. Verrà richiesto di inserire un nome al progetto (figura 3.6): per rimanere nello stesso linguaggio usato dai plugin di esempi generici in Eclipse, verrà messo il nome *com.example.helloworld*.

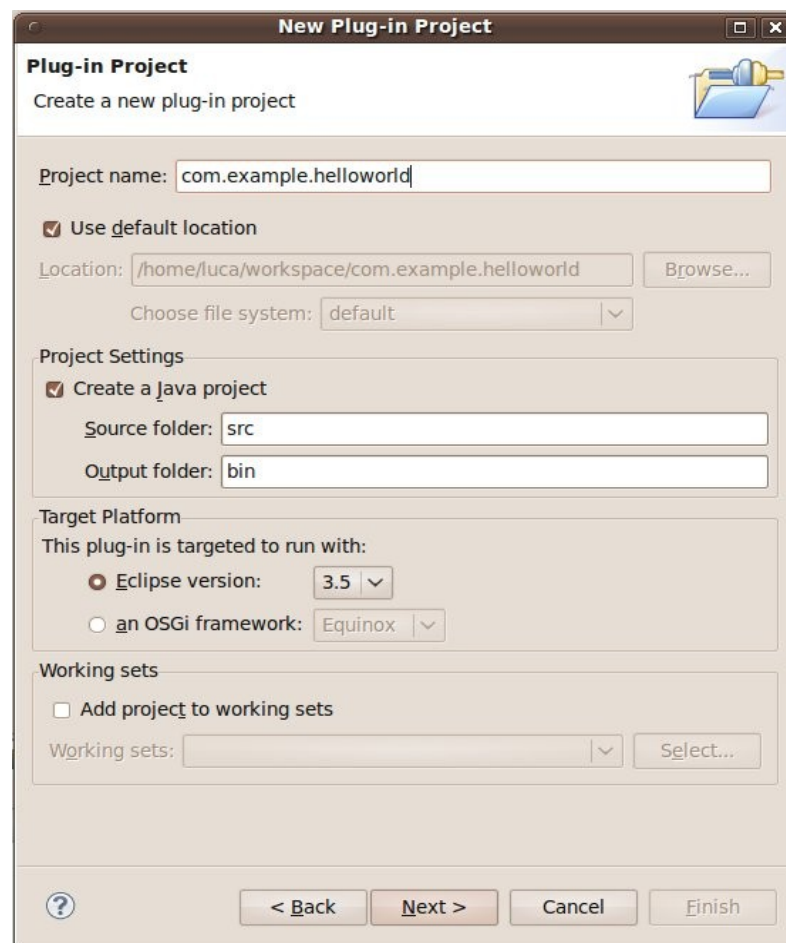


Figura 3.6: Fase I della creazione plugin esempio

Procedendo nella creazione guidata, verranno richieste altre informazioni (figura 3.7), quali l'ID, la versione e il fornitore: preferibilmente è meglio inserire lo stesso nome sia nell'ID del

plugin che nel nome del progetto, in quanto si avranno meno problemi nel caso esista già un plugin con lo stesso nome. Si noti che in questa videata vengono selezionate la spunta per generare automaticamente il codice per l'*Activator.java*, ovvero quello che si occuperà dell'attivazione del plugin, e la spunta per eseguire il nuovo plugin nella interfaccia-utente (ovvero il *Workbench*).

New Plug-in Project

Content
Enter the data required to generate the plug-in.

Properties

ID:

Version:

Name:

Provider:

Execution Environment:

Options

☒ Generate an activator, a Java class that controls the plug-in's life cycle
Activator:

☒ This plug-in will make contributions to the UI

☐ Enable API Analysis

Rich Client Application
Would you like to create a rich client application? ☐ Yes ☒ No

Figura 3.7: Fase II della creazione plugin esempio

Nella fase successiva selezionare, tra i modelli forniti, **Hello, World** (figura 3.8), permettendo così la generazione automatica del codice del nuovo plugin basato su quello standard, potendolo poi

modificare in seguito.

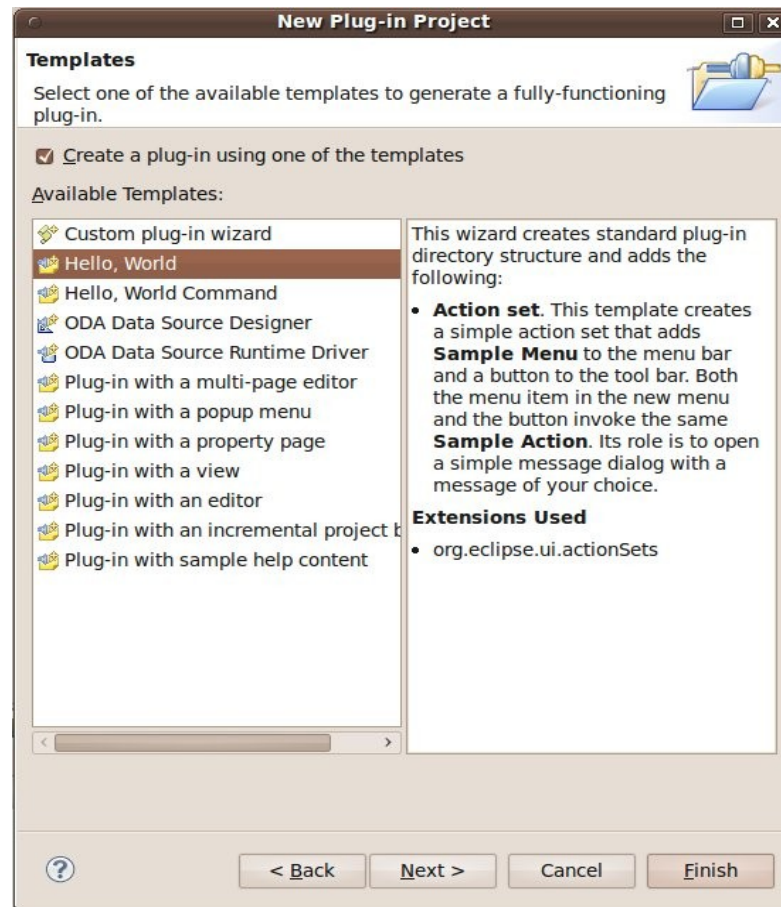


Figura 3.8: Fase III della creazione plugin esempio

Procedendo, viene ora mostrato le classi che utilizzerà il nostro plugin per l'esecuzione (figura 3.9): in questo caso si utilizzerà la classe *SampleAction* attraverso l'*extension*, che permetterà il collegamento con il plugin *org.eclipse.ui.actionSets*, creandola nel nuovo *Java package* *com.example.helloworld.actions* e inserendo il messaggio "Hello World!". Tale collegamento darà la possibilità di creare una voce all'interno del menù di Eclipse.

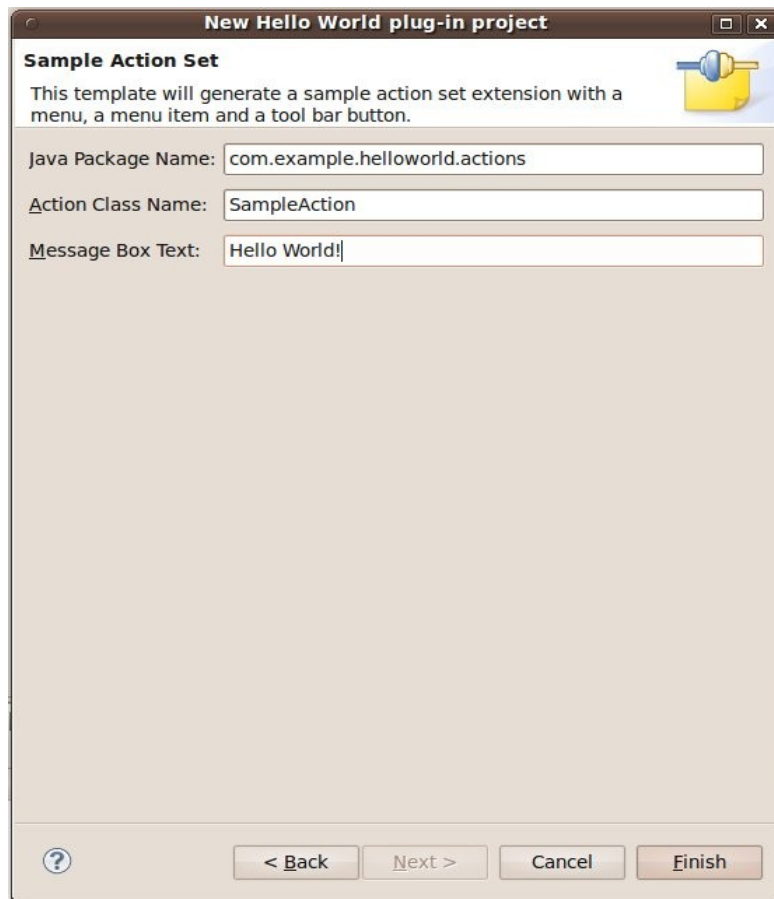


Figura 3.9: Fase IV della creazione plugin esempio

Finita la procedura guidata, si avrà la *directory* *com.example.helloworld* all'interno del proprio *Workspace*, notando tale nome nella *perspective* del *Workbench* in uso (figura 3.10), in particolare nel *Package Explorer*.

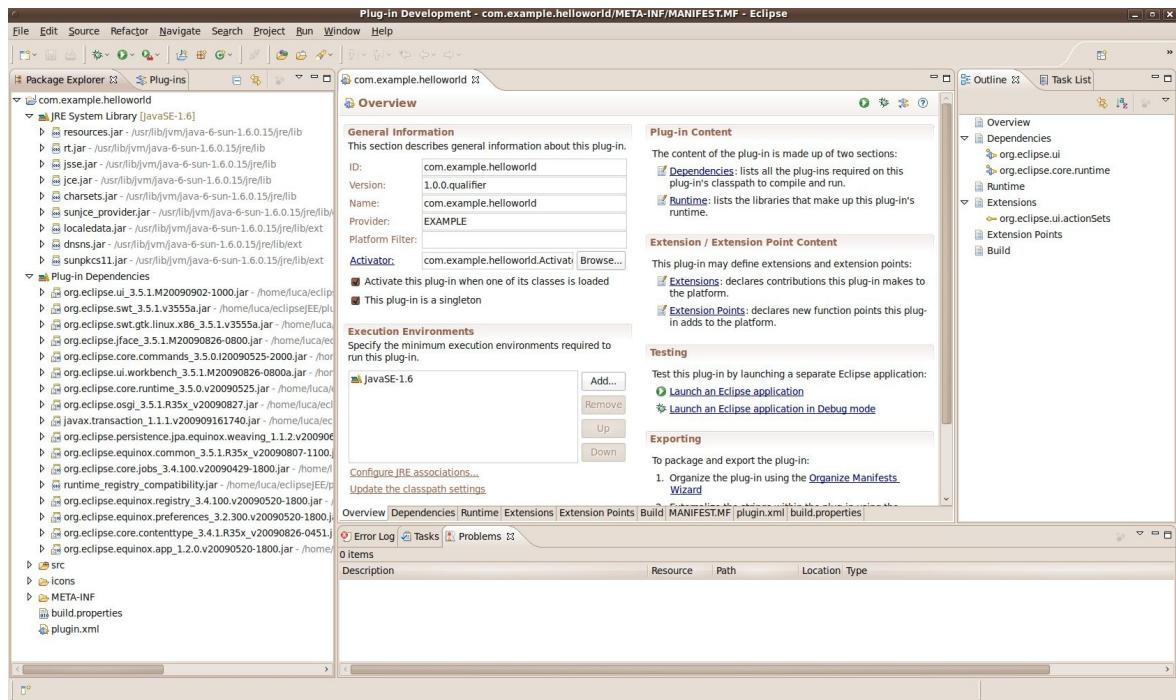


Figura 3.10: Veduta d'insieme del plugin esempio “Hello World” nella prospettiva di PDE

Nella veduta d'insieme del *Package Explorer*, sono presenti le parti create automaticamente dalla procedura guidata appena eseguita; tra le parti “meno rilevanti” vi sono i file *JAR* che includono le classi necessarie di Eclipse e di Java per l'esecuzione del plugin, una cartella contenente le icone (*icons*) usate dal plugin (in questo caso ce n'è solo una), e il file *build.properties* che contiene le variabili utilizzate da uno *script* costruito automaticamente. Le parti più importanti risiedono nella cartella *src* (*source*), la quale, appunto, contiene il codice sorgente per il nuovo plugin e il *plugin manifest file* *plugin.xml*.

Prima di andare ad analizzare tale file, si va a modificare le *extension*: selezionando il *tab* relativo ad esse nella videata generale del nuovo plugin, si nota il collegamento con *org.eclipse.ui.actionSets* precedentemente stabilito; selezionando la classe *Sample Menu*, si va a cambiare il nome della relativa etichetta

in “Hello World!”; stessa cosa la si fa per la classe *Sample Action*, inserendo il nome “Display Hello World!” nell'etichetta e nel *tooltip*.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>

  <extension
    point="org.eclipse.ui.actionSets">
    <actionSet
      label="Sample Action Set"
      visible="true"
      id="com.example.helloworld.actionSet">
      <menu
        label="Hello World!"
        id="sampleMenu">
        <separator
          name="sampleGroup">
        </separator>
        </menu>
        <action
          label="Display Hello World!"
          icon="icons/sample.gif"
          class="com.example.helloworld.actions.SampleAction"
          tooltip="Display Hello World!"
          menubarPath="sampleMenu/sampleGroup"
          toolbarPath="sampleGroup"
          id="com.example.helloworld.actions.SampleAction">
        </action>
      </actionSet>
    </extension>
  </plugin>
```

Figura 3.11: Plugin.xml del plugin esempio

Analizzando il file *manifest*, i primi due tag sono relativi alla versione XML in cui è stato scritto il file, la codifica dei caratteri e la versione di Eclipse. In seguito vengono scritti i tag relativi alle caratteristiche del plugin, in particolare delle sue *extension*: la voce che si riferisce alla classe a cui si vuole collegare e il tag del set delle azioni da intraprendere, ovvero la creazione di una voce nel menù denominata “Hello World!” e dell'azione “Display Hello World!” correlata ad essa, comprendendo le relative relazioni alle classi di Eclipse.

Viene così generato automaticamente il codice sorgente Java del nuovo plugin nei file *SampleAction.java* (figura 3.12) e *Activator.java* (figura 3.13), presenti nella cartella *src* del *Package*

Explorer: si noterà la presenza delle importazioni, delle implementazioni e della definizione di variabili, costruttori, funzioni e procedure, già compilate automaticamente grazie alla procedura guidata del *PDE* appena eseguita finora.

```
package com.example.helloworld.actions;

import org.eclipse.jface.action.IAction;

/**
 * Our sample action implements workbench action delegate.
 * The action proxy will be created by the workbench and
 * shown in the UI. When the user tries to use the action,
 * this delegate will be created and execution will be
 * delegated to it.
 * @see IWorkbenchWindowActionDelegate
 */
public class SampleAction implements IWorkbenchWindowActionDelegate {
    private IWorkbenchWindow window;

    /**
     * The constructor.
     */
    public SampleAction() {
    }

    /**
     * The action has been activated. The argument of the
     * method represents the 'real' action sitting
     * in the workbench UI.
     * @see IWorkbenchWindowActionDelegate#run
     */
    public void run(IAction action) {
        MessageDialog.openInformation(
            window.getShell(),
            "com.example.helloworld",
            "Hello World!");
    }

    /**
     * Selection in the workbench has been changed. We
     * can change the state of the 'real' action here
     * if we want, but this can only happen after
     * the delegate has been created.
     * @see IWorkbenchWindowActionDelegate#selectionChanged
     */
    public void selectionChanged(IAction action, ISelection selection) {
    }

    /**
     * We can use this method to dispose of any system
     * resources we previously allocated.
     * @see IWorkbenchWindowActionDelegate#dispose
     */
    public void dispose() {
    }

    /**
     * We will cache window object in order to
     * be able to provide parent shell for the message dialog.
     * @see IWorkbenchWindowActionDelegate#init
     */
    public void init(IWorkbenchWindow window) {
        this.window = window;
    }
}
```

Figura 3.12: SampleAction.java

```

package com.example.helloworld;

import org.eclipse.jface.resource.ImageDescriptor;

/**
 * The activator class controls the plug-in life cycle
 */
public class Activator extends AbstractUIPlugin {

    // The plug-in ID
    public static final String PLUGIN_ID = "com.example.helloworld";

    // The shared instance
    private static Activator plugin;

    /**
     * The constructor
     */
    public Activator() {
    }

    /**
     * (non-Javadoc)
     * @see org.eclipse.ui.plugin.AbstractUIPlugin#start(org.osgi.framework.BundleContext)
     */
    public void start(BundleContext context) throws Exception {
        super.start(context);
        plugin = this;
    }

    /**
     * (non-Javadoc)
     * @see org.eclipse.ui.plugin.AbstractUIPlugin#stop(org.osgi.framework.BundleContext)
     */
    public void stop(BundleContext context) throws Exception {
        plugin = null;
        super.stop(context);
    }

    /**
     * Returns the shared instance
     *
     * @return the shared instance
     */
    public static Activator getDefault() {
        return plugin;
    }

    /**
     * Returns an image descriptor for the image file at the given
     * plug-in relative path
     *
     * @param path the path
     * @return the image descriptor
     */
    public static ImageDescriptor getImageDescriptor(String path) {
        return imageDescriptorFromPlugin(PLUGIN_ID, path);
    }
}

```

Figura 3.13: Activator.java

Il nuovo plugin è ora pronto per essere eseguito: per farlo, basta andare nella tab “Overview” del plugin stesso e selezionare “Launch an Eclipse application”. In questo modo si aprirà una nuova istanza della piattaforma Eclipse, nella quale sarà presente il nostro plugin.

Si noterà ora la presenza di una nuova voce nel menù, ovvero “Hello World!”. Per eseguire l'azione del plugin, basta andare su **Hello World!->Display Hello World!** e comparirà la finestra (figura 3.14) denominata “*com.example.helloworld*”, con il logo scelto all'interno della cartella *icons* e la frase inserita “Hello World!”.

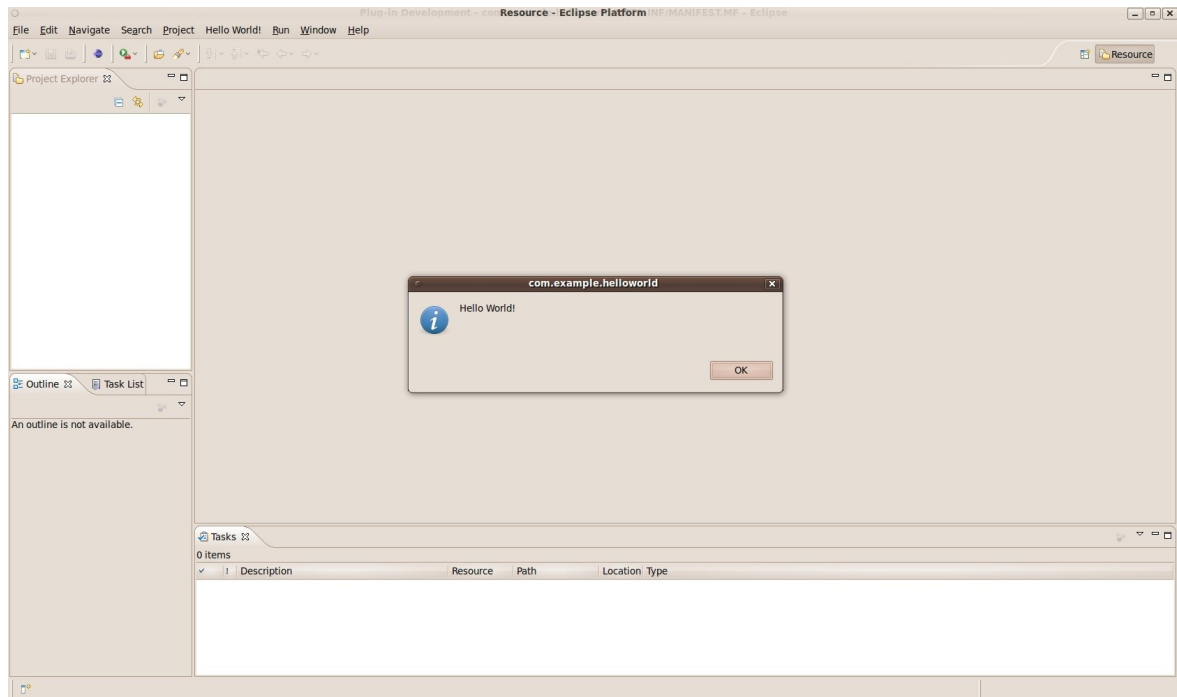


Figura 3.14: Esecuzione del plugin esempio

4. Analisi del sistema

In questo capitolo vengono prese in considerazione le specifiche del sistema che si andrà a realizzare; viene poi eseguita l'analisi dei requisiti e di seguito i casi d'uso e gli scenari.

4.1 Specifiche del sistema

Viene richiesta la realizzazione di un tool (*Graph&SODA*) che dia supporto al linguaggio grafico della metodologia SODA; dovrà supportare in particolare il progettista durante tutte le fasi dello sviluppo attraverso la modellazione grafica del sistema.

Graph&SODA è un IDE (*Integrated Development Environment*) grafico, realizzato attraverso plugin di Eclipse, che dovrà inoltre essere facilmente integrabile con altri plugin che saranno sviluppati in seguito; in particolare dovrà integrarsi con *Check&SODA* e *Kit&SODA*, i quali rispettivamente dovranno occuparsi della consistenza del sistema e del suo completo sviluppo, curando la gestione e l'integrazione degli altri plugin appena citati.

Verrà data la possibilità di usare tale metodologia per lo sviluppo del sistema sia attraverso gli input dell'utente, sia attraverso la ricezione delle tabelle già precompilate.

Sarà posta particolare attenzione ai meccanismi fondamentali utilizzati nel processo SODA, quali il *layering* ed il *carving*, rendendo pratico il più possibile il loro utilizzo da parte dell'utente, offrendo una visione globale dei vari livelli di astrazione o una visione di ognuno di essi.

4.2 Analisi dei Requisiti del sistema

All'utente viene data la possibilità di “inserire” e creare le entità astratte per ogni fase del processo SODA, partendo dal livello 0 o *Core layer*. Partendo dalla prima fase di Analisi dei Requisiti, viene eseguita la creazione di attori, di requisiti, di *legacy-systems* e di relazioni tra le entità, dando a ciascuno di questi una descrizione: si ottiene così la creazione di parte delle prime tabelle relazionali di questa fase (rispettivamente $(0)Ac_i$, $(0)Re_i$, $(0)LS_i$, $(0)Rel_i$). In seguito, per completare la stesura delle restanti tabelle appartenenti a questa fase, l'utente dovrà inserire le relazioni tra le vari entità presenti nello stesso livello (spiegato nel dettaglio di seguito nella parte della vista delle tabelle), nello specifico quelle relative alle tabelle delle tipologie che le richiedono; in questo caso, essendo nel livello 0 della fase di Analisi dei Requisiti, tali tabelle sono: $(0)AR_i$ per quanto riguarda le Tabelle dei Requisiti (*Requirement Tables*); $(0)EELS_i$ per le Tabelle di Dominio (*Domain Tables*); $(0)RR_i$ e $(0)LSR_i$ per le Tabelle delle Relazioni (*Relations Tables*).

A questo punto si è già in grado di dare una visione di una prima iterazione di questa fase. In particolare si possono avere due viste per ogni singola fase: la prima mostra tutte le entità astratte nel loro insieme (*Abstractions View*, esempio in fase di Analisi dei Requisiti in figura 4.1), la seconda le tabelle relazionali (*Tables View*, esempio in fase di Analisi dei Requisiti in figura 4.2). Tali viste possono essere sviluppate per tutte le fasi, dando la possibilità all'utente di decidere di procedere alla fase successiva (una volta terminata quella in cui stava lavorando) o di tornare indietro (ristabilendo nuovamente diverse entità e loro relazioni). Un'eccezione, però, viene introdotta per le fasi di passaggio: le entità inserite in queste fasi sono solo relazioni con le entità create nella precedente fase, quindi non avranno visione nella *Abstractions*

View, in quanto si creerebbe un controsenso nel rappresentare nella stessa vista entità appartenenti a diverse fasi (seppur situate nello stesso livello di astrazione), bensì saranno visibili solo come tabelle nella *Tables View*.

Prima di andare a vedere le viste nel dettaglio, viene introdotto un principio fondamentale nel processo SODA: il *layering*. Secondo la “volontà” dell'utente, si può iterare ogni fase (eccetto quella di Design di Dettaglio, per definizione) attraverso il *layering* delle entità che lo necessitano (selezionando semplicemente l'entità e mostrando lo sviluppo di tale operazione), dando origine così ad altre entità (proiettate o zoomate) in vari livelli di astrazione e all'aggiunta della tabella di Zoom (*Zooming Table*, $(L)Z_t$) presente per ogni *layer* più o meno dettagliato.

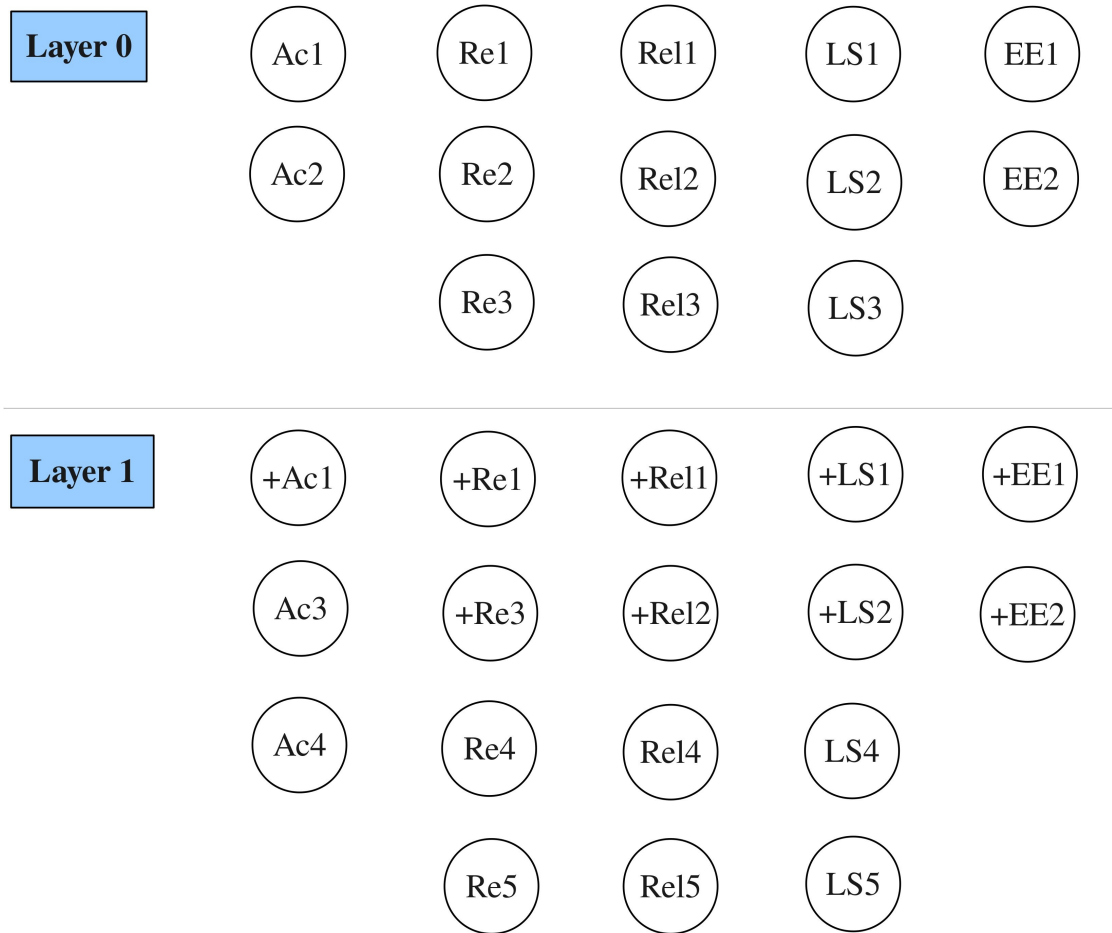
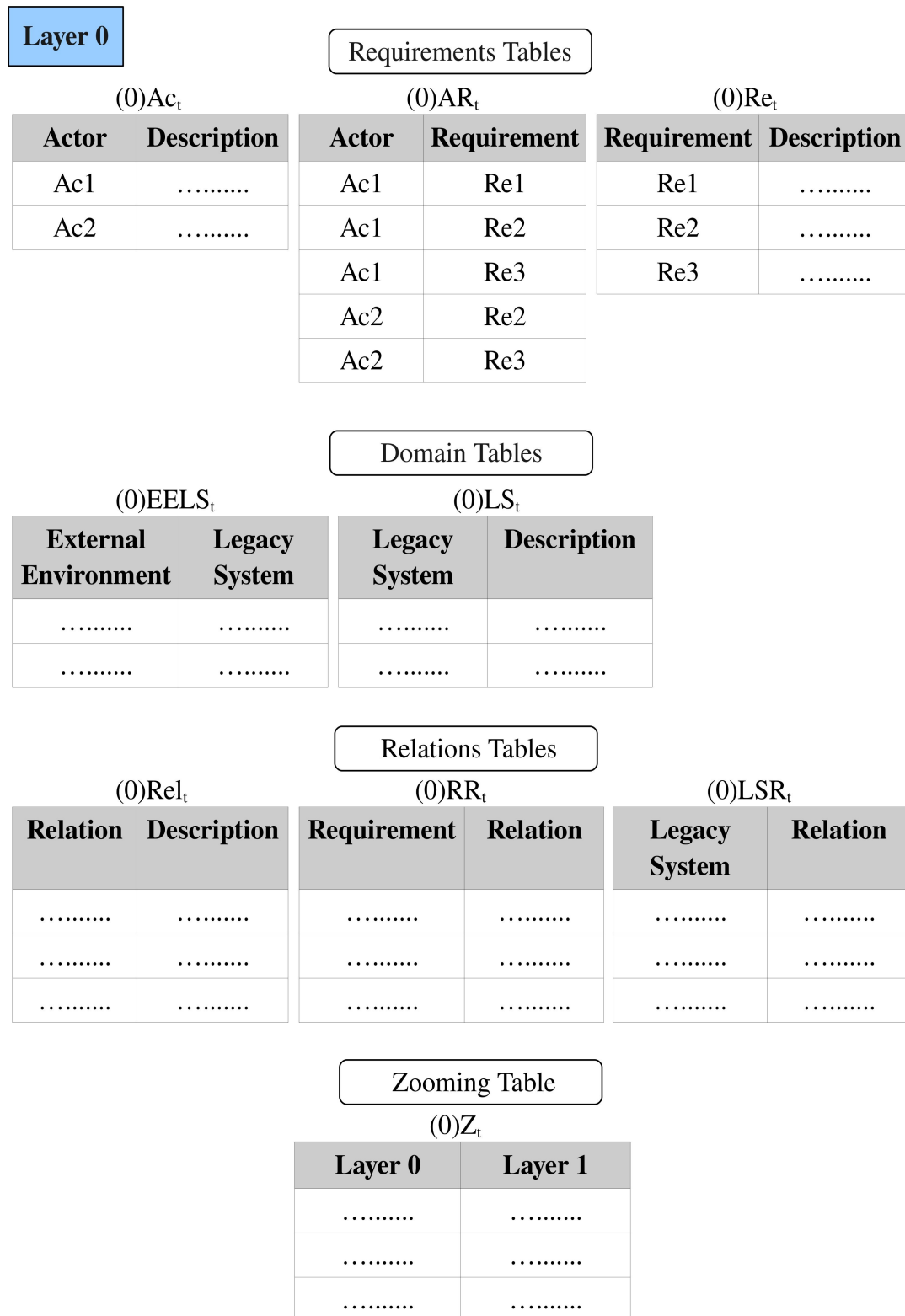


Figura 4.1: Abstractions View

Nella *Abstractions View* vengono visualizzate le singole astrazioni presenti nei vari livelli, quelle proiettate o zoomate. In particolare viene data la possibilità di vedere i singoli livelli o la visione di insieme di tutti i livelli (utile soprattutto per il *carving* nella fase successiva al Design Architettuale), mostrando le entità che vi fanno parte. Sarebbe anche possibile illustrare le varie relazioni presenti tra le entità attraverso frecce che le collegano, ma risulterebbe molto confusionario, soprattutto in MAS molto complessi e voluminosi. Viene data poi la possibilità di andare a visualizzare e modificare le tabelle, selezionando la singola entità (mostrando le tabelle in cui tale entità è coinvolta, compresa la *Zooming Table*).



Analogamente viene rappresentato il layer 1, cambiando la numerazione delle tabelle relative ad esso.

Figura 4.2: Tables View

Nella *Tables View* vengono visualizzate le tabelle della fase in questione, suddivise per livello e per tipologia di ogni fase del processo SODA (in questo caso *Requirement Tables*, *Domain Tables*, *Relations Tables*, *Zooming Table*).

Come detto in precedenza, la creazione di tabelle per ogni fase avviene mediante gli input dell'utente. In particolare, una volta create le tabelle di ogni singola fase relative alla creazione delle singole entità, verranno generate quelle successive che gestiscono le interrelazioni presenti tra tali astrazioni, attraverso l'associazione di ciascuna entità con quella relazionata ad ognuna di esse. Per quanto riguarda il passaggio da una fase all'altra, si andranno a creare le tabelle dall'associazione delle nuove entità della nuova fase con quelle della fase precedentemente completata, sempre suddivise per livello di appartenenza, a parte le Tabelle di *Mapping* e quelle della fase di Design di Dettaglio che non hanno suddivisioni in livelli, poiché hanno già subito la scelta del miglior livello di astrazione delle fasi precedenti.

L'altro meccanismo fondamentale utilizzato nel processo SODA è il *carving*, che avviene subito prima della stesura delle Tabelle di *Mapping*, ovvero tra il passaggio dalla fase di Analisi Architettuale a quella di Design di Dettaglio. Tale meccanismo verrà eseguito semplicemente selezionando (nella *Abstractions View*) le entità che si vogliono “ritagliare” dai vari livelli per mapparle poi per il Design di Dettaglio. Si ricorda nuovamente che dopo il *carving* non si parlerà più di livelli di astrazione e le tabelle non si riferiranno più a loro.

Viene data così la possibilità di sviluppare graficamente il processo SODA attraverso i passaggi di ogni singola fase, partendo dagli input dell'utente. Ma non sarà l'unico modo: viene data la possibilità di realizzare tale grafica anche inserendo le tabelle già precompilate

dall'utente, dandogli così la possibilità di vedere graficamente ciò che ha progettato, di controllare, migliorare e continuare a sviluppare il sistema da lui stesso progettato. L'utente può inserire tali tabelle sia nella prima fase del processo SODA (Analisi dei Requisiti), sia in quelle successive, avendo sempre la possibilità di correggere dei dati e di continuare la progettazione anche dove non abbia le tabelle già precompilate. Tali tabelle vengono considerate come oggetti, distinti per la loro tipologia: in particolare verranno sviluppati in linguaggio JAVA, restando così nella stessa ottica in cui è sviluppato il tool in Eclipse. Lo scambio dei dati avverrà attraverso l'utilizzo di file XML, vista l'alta personalizzabilità, comprensione e compatibilità.

Graph&SODA, oltre all'importazione, metterà a disposizione anche l'esportazione dei dati relativi alle tabelle, per permettere così l'archiviazione di tali dati ed il loro utilizzo da altri programmi.

Per non appesantire il tool grafico, la coerenza di tali tabelle verrà constatata dal tool *Check&SODA*, ancora non presente: *Graph&SODA* metterà a disposizione l'interfacciamento che deve avere il tool per interagire con esso (e non solo) e viceversa, rispettivamente attraverso gli *extension point* e le *extension*, messi a disposizione da Eclipse per il collegamento tra i vari plugin (come detto nel capitolo 3). Per ora verranno eseguite prove con l'inserimento di tabelle “sicure”.

4.3 Casi d'uso e scenari

Per avere una maggior concretezza dell'utilizzo del tool *Graph&SODA* fin qui analizzato, verranno ora descritte le principali attività che un utente può eseguire utilizzando il tool grafico in questione.

4.3.1 Creazione delle entità astratte

L'utente può richiedere la creazione di un'entità astratta in una determinata fase del processo SODA (ad eccezione di quelle di passaggio), ove gli viene richiesto il nome di tale entità (che dovrà avere un nome diverso dalle altre già presenti) e la propria descrizione. Verrà così aggiunta quella nuova nel livello di riferimento nella *Abstractions View* e la nuova voce nella relativa tabella del livello di riferimento, mostrata nella *Tables View*.

Per le fasi di passaggio, invece, nella creazione di una nuova entità astratta viene richiesto all'utente di specificare a quale entità della fase precedente sia relazionata, permettendo così solo la creazione delle tabelle visibili nella *Tables View*.

4.3.2 Gestione delle relazioni tra le varie entità astratte

Oltre all'inserimento delle singole entità, per completare la stesura delle tabelle, l'utente deve stabilire i rapporti che intercorrono tra le varie entità da lui introdotte, permettendogli l'associazione di ogni singola entità con altre precedentemente create, andando a stabilire così le tabelle che completano lo sviluppo della fase.

4.3.3 Inserimento tabelle già precompilate

L'utente può essere già in possesso delle tabelle (relative alla fase in cui si trova) già precompilate; ha quindi la possibilità di inserirle nella sua progettazione attraverso l'importazione dei dati presenti all'interno del file XML che descrive ed elenca tali tabelle. Il

controllo della consistenza di tali dati viene eseguito tramite l'utilizzo di plugin esterni (vedi paragrafo 4.3.10), grazie alla estendibilità del tool grafico. L'utente sarà anche in grado di modificare le suddette tabelle appena inserite, così come di aggiungere ed integrare altre nuove entità.

4.3.4 Layering

Per ogni fase del processo SODA, l'utente può eseguire il *layering* delle entità che lo necessitano. Tale meccanismo avviene specificando la volontà dell'utente di eseguirlo e selezionando (nella *Abstractions View*) l'entità in questione; a questo punto verranno aggiunti un nuovo livello (più o meno dettagliato), le nuove entità derivanti dal *layering*, le nuove tabelle del livello appena inserito e la *Zooming Table*. Verranno poi create anche le restanti tabelle che gestiscono le relazioni tra le nuove entità, così come si è parlato nel paragrafo 4.3.2.

4.3.5 Carving

Ad un passo prima dell'ultima fase, viene eseguito il *carving*: l'utente ha la necessità di scegliere le “migliori” entità nei vari livelli; in particolare andrà a “ritagliare”, secondo la sua volontà, il sistema che meglio rappresenterà il suo progetto, semplicemente selezionando le entità nella *Abstractions View*. Da tale meccanismo, l'utente passerà ad elencare le tabelle di *Mapping* che precedono l'ultima fase di Design di Dettaglio, avendone visione solo nella *Tables View* (essendo un passaggio tra due fasi, vedi paragrafo 4.3.1).

4.3.6 Esportazione tabelle

Come già detto nel paragrafo 4.2, l'utente può richiedere ed effettuare l'esportazione delle tabelle da lui create, attraverso il loro salvataggio in un file XML, permettendogli così l'archiviazione e l'eventuale integrazione con programmi esterni.

4.3.7 Vista delle entità astratte (*Abstractions View*)

Uno degli obiettivi principali è appunto la visione delle entità che l'utente stesso ha creato (vedi paragrafo 4.3.1) o importato (vedi paragrafo 4.3.3); riesce così ad averla livello per livello, oppure completa di tutti i livelli (utile soprattutto per il *carving*).

4.3.8 Vista delle tabelle (*Tables View*)

L'altra importante vista è quella delle tabelle: l'utente è in grado di vederle, anche nel loro dettaglio, suddivise per tipologia di ogni fase e per livello, avendo sempre la loro visione di insieme. Selezionando una tabella, ha inoltre la possibilità di modificare le entità e le loro relazioni.

4.3.9 Visualizzazione delle tabelle di cui fa parte una specifica entità

Viene dato all'utente anche un approccio rapido per la modifica e visualizzazione di una singola entità e delle relazioni in cui è

coinvolta: semplicemente selezionando un'entità nella *Abstractions View*, vengono visualizzate le tabelle (nella struttura della *Tables View*) relative all'entità scelta nello specifico livello di appartenenza, permettendo sempre l'eventuale modifica (vedi paragrafo precedente).

4.3.10 Integrazione con altri plugin

Oltre all'esportazione delle tabelle (paragrafo 4.3.6), il tool grafico avrà la possibilità di interfacciarsi con altri plugin, permettendo così il suo utilizzo anche al di fuori di se stesso e, a sua volta, di poter utilizzare altre funzioni esterne (come descritto nel paragrafo 4.2): questo permette la sua semplice funzionalità ed espandibilità (uno degli obiettivi principali di Eclipse). In futuro sarà possibile integrare anche collegamenti diretti verso funzionalità esterne, come ad esempio la possibilità di controllare la consistenza del sistema in progettazione attraverso il tool *Check&SODA* con un semplice “click”.

4.4 Schema logico di riferimento

Dalle considerazioni fatte precedentemente nell'analisi dei requisiti ed essendo noti i casi d'uso, nel sistema che si andrà a sviluppare vengono identificate le seguenti entità:

- Entità Astratta (*Abstraction*): rappresenta l'entità che costituisce l'astrazione presente nelle varie fasi del processo SODA, comprendendo anche quella proiettata o zoomata, derivata dal *layering*; ogni fase avrà il suo set di astrazioni

possibili (ad esempio, nella fase di Analisi dei Requisiti, *Actor*, *Requirement*, *Legacy-System*, *External-Environment*, *Relation*).

- Tabella (*Table*): rappresenta la tabella contenente le descrizioni o relazioni delle varie astrazioni; ogni livello ha il suo set di tabelle possibili, suddivise per le loro categorie e comprendendo anche la *Zooming Table*.
- Livello (*Layer*): rappresenta il livello di riferimento per le varie astrazioni e tabelle esistenti; partendo da quello iniziale, il livello 0 o *Core Layer*, può crescere o diminuire attraverso il *layering*.

Le decisioni delle loro creazioni vengono prese dall'utente, utilizzando il tool grafico; in particolare sarà l'utente stesso a descrivere le entità, le tabelle, a stabilire i *layer* (attraverso il principio di *layering*) e ad eseguire il *carving*; ma, per quanto riguarda l'interfacciamento grafico e le operazioni da eseguire, sarà il tool grafico a gestire le viste delle varie fasi di SODA (*Abstractions View* e *Tables View*), attraverso l'interazione tra i vari plugin di Eclipse. Nella figura 4.3 viene riportata una veduta d'insieme dell'interazione tra i vari componenti.

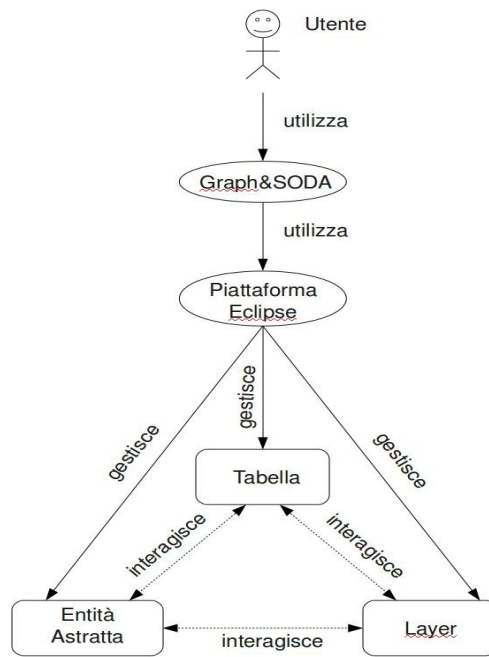


Figura 4.3: Overview delle interazioni tra i componenti

5. Progettazione ed Implementazione del sistema

5.1 Struttura delle entità

Dallo schema logico mostrato in sede di analisi, oltre all'utente e alle tre entità individuate (*Abstraction*, *Table*, *Layer*), sono presenti due “macro-parti” (“Graph&SODA” e “Piattaforma Eclipse”) che faranno da intermediari tra l'utente e le entità stesse: l'interazione tra tali parti può complessivamente essere rappresentata dalle due viste (*Abstractions View* e *Tables View*), in quanto, per l'utente, sono l'interfacciamento per lo sviluppo del proprio sistema. Si può, così, definire così il diagramma UML (figura 5.1):

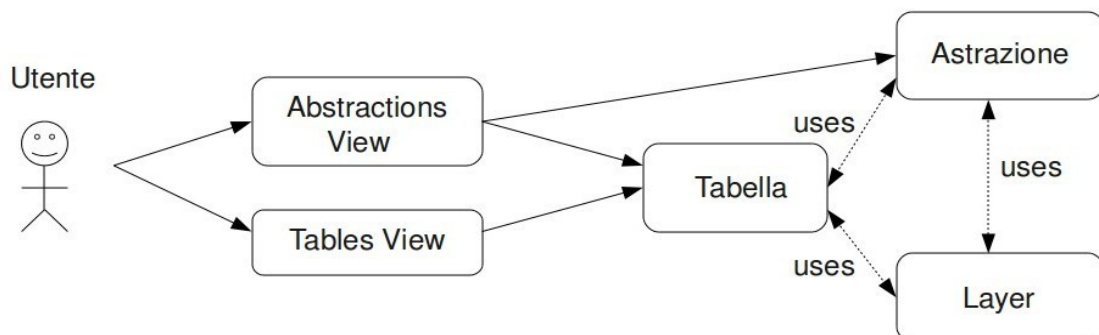


Figura 5.1: Diagramma UML

Nella figura 5.1, l'utente può interagire esclusivamente con le due viste, attraverso le quali può gestire la progettazione del suo sistema, procedendo con la creazione di astrazioni e tabelle: tali entità si scambiano dati al fine di realizzare le viste.

Verrà ora definita la struttura delle entità, analizzandole singolarmente.

5.1.1 Astrazione

Ogni astrazione (*Abstraction*) può essere vista come una classe Java (figura 5.2), caratterizzata dal nome, dalla tipologia (ad esempio *Actor*, *Requirement*), dalla fase del processo SODA appartenente e dal livello in cui è presente (fatta eccezione per le astrazioni dopo il *carving*).

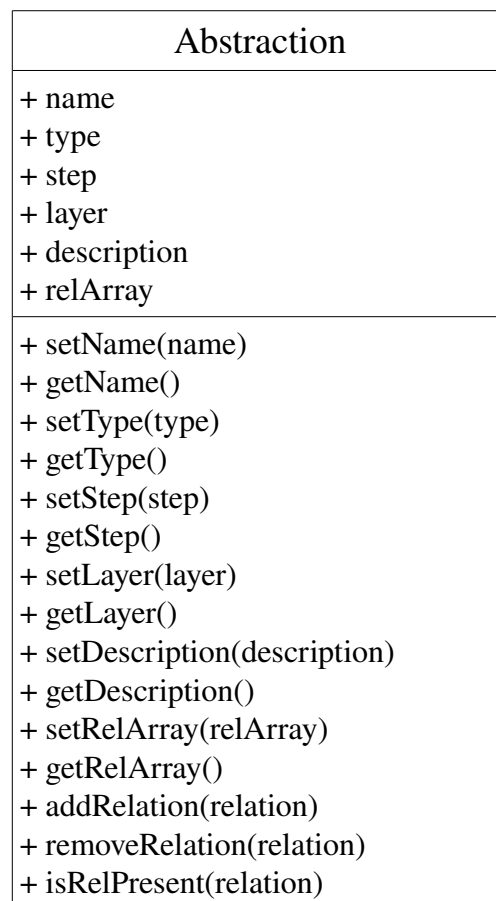


Figura 5.2: Classe *Abstraction* in UML

Le descrizioni delle sue funzioni e procedure, pertanto, sono le seguenti:

- *setName(name)* e *getName()*: per settare o restituire il nome;
- *setType(type)* e *getType()*: per settare o restituire la tipologia;
- *setStep(step)* e *getStep()*: per settare o restituire la fase in cui si trova;
- *setLayer(layer)* e *getLayer()*: per settare o restituire il layer a cui appartiene.

Per praticità e per permettere la miglior interazione tra le entità, viene inclusa nella classe anche la sua descrizione, avendo così *setDescription(description)* e *getDescription()* per settare o restituire la sua descrizione.

All'interno della classe sono presenti anche le relazioni tra le varie astrazioni; in particolare, per gestirle si hanno i seguenti metodi:

- *setRelArray(relArray)* e *getRelArray()*: per settare o restituire l'elenco delle relazioni (*relation*) tra le varie astrazioni;
- *addRelation(relation)* e *removeRelation(relation)*: per aggiungere o rimuovere una relazione nell'elenco;
- *isRelPresent(relation)*: per informare della presenza di una relazione nell'elenco.

5.1.2 Tabella

L'entità *Table* è incapsulata in una classe Java (figura 5.3), il quale è caratterizzato dal nome, dalla tipologia, dall'elenco delle relazioni tra le varie entità o le loro descrizioni (con relazioni si intendono gli accostamenti), e dal livello a cui si riferisce (fatta eccezione per quelle dopo il *carving*).



Figura 5.3: Classe *Table* in UML

Ecco le sue funzioni e procedure nel dettaglio:

- *setName(name)* e *getName()*: per settare o restituire il nome;
- *setType(type)* e *getType()*: per settare o restituire la tipologia;
- *setRelArray(relArray)* e *getRelArray()*: per settare o restituire l'elenco delle relazioni (*relation*) tra le varie entità o le loro descrizioni;

- *setLayer(layer)* e *getLayer()*: per settare o restituire il layer a cui appartiene.

Anche qui, per praticità di individuazione, viene inclusa nell'oggetto la fase del processo SODA di cui fa parte, inserendo così *setStep(Step)* e *getStep()* per settare o restituire la fase a cui appartiene.

5.1.3 Layer

L'entità *Layer* è rappresentata da una classe Java (figura 5.4), caratterizzata dal numero e dalla fase a cui fa riferimento, dalle entità che in esso sono coinvolte e dalle tabelle presenti.

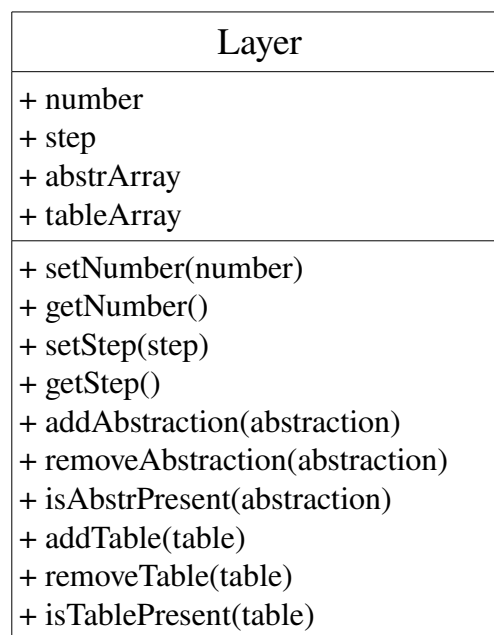


Figura 5.4: Classe *Layer* in UML

I dettagli delle sue funzioni e procedure sono:

- *setNumber(number)* e *getNumber()*: per settare o restituire il numero del livello che rappresenta;
- *setStep(step)* e *getStep()*: per settare o restituire la fase a cui appartiene;
- *addAbstraction(abstraction)* e *removeAbstraction(abstraction)*: per aggiungere o rimuovere un'entità presente in esso;
- *isAbstrPresent(abstraction)*: per informare della presenza di una astrazione nell'elenco;
- *addTable(table)* e *removeTable(table)*: per aggiungere o rimuovere una tabella presente in esso;
- *isTablePresent(table)*: per informare della presenza di una tabella nell'elenco.

5.2 Le viste

Le classi più complicate del diagramma UML esaminato, sono le due viste: *Abstractions View* e *Tables View*. Esse dovranno essere in grado di visualizzare e modificare le entità e le tabelle del sistema che l'utente sta sviluppando, tenendo conto delle regole e dei principi di SODA.

Tali viste verranno ora progettate, mostrando il loro funzionamento nel dettaglio.

5.2.1 *Abstractions View*

Questa vista è indispensabile per la visione d'insieme del sistema che si vuole realizzare; in particolare rappresenta un'area dove si permette la visualizzazione (e la modifica) di tutte le astrazioni presenti nello specifico livello (o di tutte le astrazioni di tutti i livelli) di una determinata fase del processo SODA. L'utente ha quindi la necessità di avere una barra di strumenti dove, in primis, può scegliere le astrazioni che può inserire in quel livello: dopo aver chiesto all'utente di inserire il nome, la sua tipologia e la sua descrizione, tale astrazione verrà così creata e poi rappresentata nella vista. Tale procedimento è reso possibile dalla interazione tra la vista e le classi definite precedentemente; i passaggi di tali operazioni sono i seguenti: viene creato un oggetto *Abstraction*; se la sua creazione è andata a buon fine, si procede con la creazione della tabella (ovvero l'oggetto *Table*) relativa alla sua tipologia, a meno che non sia già esistente; si procede, in seguito, con la sua aggiunta in tale tabella; avviene poi l'inserimento di tale tabella e dell'astrazione nell'oggetto *Layer* (già presente al momento dell'inizializzazione della fase).

Altre necessità dell'utente (oltre ai meccanismi fondamentali di cui si parlerà nei paragrafi successivi) sono il passaggio da una fase ad un'altra del processo SODA e la scelta della vista; vengono inseriti dei pulsanti, in entrambe le viste, per la loro gestione.

Il passaggio “indotto” (ovvero non scelto direttamente dall'utente) tra le due viste avviene quando l'utente desidera vedere le tabelle in cui è coinvolta una determinata entità: attivando il pulsante relativo a tale scopo, l'utente seleziona l'astrazione di cui vuole vedere tutte le tabelle ove è presente, passando così alla vista *Tables View*, mostrando solo tali tabelle.

5.2.1.1 Layering

Un aspetto fondamentale è il principio di *layering*. Per attuarlo durante le varie fasi (eccetto quella successiva al *carving*), viene messo a disposizione un pulsante nella vista; attivando tale pulsante, viene data la possibilità all'utente di selezionare l'entità (una alla volta) che necessita di essere zoomata o proiettata in un altro livello. Per quanto riguarda il meccanismo di *zoom*, i procedimenti sono i seguenti: viene richiesto se eseguire un *in-zoom* o un *out-zoom* dell'entità selezionata; se viene richiesto il primo, si creerà (se non è già presente) un nuovo livello (oggetto *Layer*) più dettagliato, nel quale verranno generati ed inseriti i nuovi oggetti *Abstraction* (derivati da tale meccanismo) e *Table* (creati dall'inserimento degli *Abstraction* appena introdotti); verrà anche creata la *Zooming Table* ed inserita nel livello di origine, con le relazioni tra l'entità di partenza e quelle generate dal meccanismo. Se viene richiesto l'*out-zoom*, invece, verrà generato (nel caso non sia già presente) un nuovo livello più astratto, nel quale verrà creato, e poi inserito, un nuovo oggetto *Abstraction* e uno *Table*; per quanto riguarda la *Zooming Table*, avviene lo stesso procedimento dell'*in-zoom*, con l'unica differenza che tale tabella viene inserita nel livello di destinazione.

Il meccanismo di *proiezione* esegue gli stessi passaggi dello *zoom* (la proiezione di una astrazione in un livello più dettagliato come l'*in-zoom*, la proiezione in un livello più astratto, come l'*out-zoom*), con la sola differenza che si andranno a creare nuovi singoli oggetti (ove necessario), uno per ogni classe.

All'atto della creazione di tutti gli oggetti (*Abstraction*, *Table* e *Layer*), viene sempre eseguito un controllo per verificare che non siano già esistenti, evitando così di creare conflitti.

Il *layering* è attuabile solo nella *Abstractions View*; pertanto il

pulsante che renderà possibile tale principio, sarà presente solo in questa vista.

5.2.1.2 Carving

Dopo la fase di Design Architetturale, viene effettuato il *carving* per “ritagliare” le astrazioni presenti nei vari livelli, scegliendo quelle ritenute più opportune. Anche tale meccanismo può essere attuato tramite l'attivazione di un pulsante, presente solo nella *Abstractions View* e in fase di Design Architetturale. Dopo aver scelto di effettuare il *carving* tramite il pulsante, l'utente deve selezionare le astrazioni che vuole “ritagliare”: può sceglierne diverse e nei diversi livelli, avendo la possibilità anche di deselezionarle in caso di errori o ripensamenti. Una volta terminato, è necessario un pulsante per notificare la fine della “operazione”. La sua pressione implica l'inizio delle associazioni tra le nuove entità delle tabelle di *Mapping* e quelle precedenti: vengono creati nuovi oggetti *Abstraction* nel passaggio dalla fase di Design Architetturale e quella di Design di Dettaglio, che saranno senza descrizione e senza livello di appartenenza; vengono poi generate le tabelle di *Mapping*, inserendo le associazioni tra questi nuovi oggetti e quelli della fase precedente. Una volta terminate tutte le tabelle, si passa all'ultima fase di Design di Dettaglio.

5.2.2 Tables View

Con questa vista si vogliono vedere nel dettaglio le tabelle caratterizzanti ogni fase e presenti nei vari livelli. Anch'essa sarà rappresentata da un'area ove sarà possibile visualizzare e modificare tutte le tabelle appartenenti ad ogni singola fase del processo SODA

(comprese quelle di passaggio), suddivise per categorie e per livelli (fatta la solita eccezione per quelle dopo il *carving*).

Per la loro creazione, nel caso si volesse utilizzare questa vista, l'utente ha la necessità di avere una barra degli strumenti ove possa inserire le tabelle relazionali contenenti le astrazioni e le loro descrizioni, oppure le loro associazioni: alla creazione di una nuova tabella, si genera un nuovo oggetto *Table*, il quale conterrà nuove astrazioni che dovranno essere create, inizializzando un oggetto *Abstraction* e inserendolo poi con la sua descrizione (o relazionandolo con un' altra astrazione) nella tabella creata; seguirà il suo inserimento nel livello (oggetto *Layer*) in cui ci si trova.

Un'altra operazione necessaria è la modifica delle tabelle. Semplicemente selezionando la singola tabella, si avrà la possibilità di selezionare l'astrazione o la descrizione da cambiare: a tale scopo, viene introdotto un pulsante che permetterà tale modifica. Questa comporterà ulteriori controlli degli oggetti cambiati, verificando la possibilità della modifica stessa e la loro ricerca nelle varie tabelle in cui sono presenti, al fine di cambiare anche lì i riferimenti.

Viene anche data la possibilità di rimuovere le singole associazioni presenti nella tabella, andando ad eliminare gli oggetti stessi e le altre relazioni in cui erano coinvolti. Sarà un'operazione molto delicata e “pericolosa”, quando la si effettua su sistemi in fasi avanzate, poiché vi è il rischio di perdere lo sviluppo fino ad allora maturato.

La *Tables View* è indispensabile per visualizzare i passaggi tra le fasi, in quanto in essi sono presenti solo relazioni e non descrizioni di nuove astrazioni.

5.3 Implementazione del sistema

Una volta determinate le classi, si passa alla loro implementazione, andandole a definire nelle rispettive classi Java in Eclipse, al fine di rendere concreto ed operativo il tool grafico Graph&SODA. Tale implementazione viene mostrata nel CD allegato.

6. Conclusioni e sviluppi futuri

Lo scopo principale di questa tesi era quello di realizzare un tool grafico a supporto della metodologia SODA. L'approccio con tale metodologia è risultato essere di basilare importanza per poter capire e supportare al meglio il processo, gli output delle singole fasi e i meccanismi propri della metodologia.

Un punto fondamentale nello sviluppo dell'elaborato è stata la scelta di Eclipse: la piattaforma è prima di tutto “libera” (OpenSource) e rappresenta un'ottima base di partenza per realizzare il tool, in quanto offre un IDE grafico evoluto; in particolare la sua struttura, formata da plugin, rende perfettamente l'idea di “compatibilità” (riguardo la sua esecuzione in diversi sistemi operativi, grazie al linguaggio Java in cui è stato compilato) ed estendibilità. Questa ultima caratteristica è il punto di forza di Eclipse, in quanto lo porta ad essere esteso indefinitamente attraverso l'uso e l'interazione tra i vari plugin.

Il tool *Graph&SODA*, essendo alla sua prima versione, potrà sicuramente essere migliorato nei suoi vari aspetti ora più “deboli”: può essere necessaria la visione di entrambe le viste in contemporanea per confrontarle e per avere una visione completa del sistema; sarebbe utile un controllo della consistenza del sistema stesso, che si potrà ottenere attraverso l'interazione di *Graph&SODA* con altri tool futuri: ad esempio si potrà sviluppare un controllore (*Check&SODA*) della consistenza delle tabelle e, quindi, del sistema realizzato adottando SODA come metodologia, oppure un toolkit completo (*Kit&SODA*) che gestisca l'interazione tra i vari plugin relativi a SODA.

7. Bibliografia

- [1] A. Omicini: **“From Objects to Agent Societies: Abstractions and Methodologies for the Engineering of Open Distributed Systems”**, Dipartimento di Elettronica, Informatica e Sistemistica (DEIS), Bologna, Italia.
- [2] B. A. Nardi: **“Context and Consciousness: Activity Theory and Human-Computer Interaction”**, MIT Press.
- [3] A. Omicini: **“SODA: Societies and Infrastructures in the Analysis and Design of Agent-Based Systems”**, LIA, Dipartimento di Elettronica, Informatica e Sistemistica (DEIS), Bologna, Italia.
- [4] A. Molesini: **“Meta-Models, Environment and Layers: Agent-Oriented Engineering of Complex Systems”**, Dipartimento di Elettronica, Informatica e Sistemistica (DEIS), Bologna, Italia.
- [5] A. Molesini, E. Nardini, E. Denti, A. Omicini: **“Situated Process Engineering for Integrating Processes from Methodologies to Infrastructures”**, Dipartimento di Elettronica, Informatica e Sistemistica (DEIS), Bologna, Italia.
- [6] A. Molesini, A. Omicini, A. Ricci, E. Denti: **“Zooming Multi-Agent Systems”**, Dipartimento di Elettronica, Informatica e Sistemistica (DEIS), Bologna, Italia.
- [7] **Eclipse.org Wiki**, l'enciclopedia libera di Eclipse, wiki.eclipse.org .

[8] **Wikipedia**, l'enciclopedia libera, it.wikipedia.org

[9] **Sito ufficiale della Eclipse Community**, www.eclipse.org

[10] **Sito ufficiale della OSGi Alliance**, www.osgi.org